A Brief Overview of the Wyscript Compiler's JavascriptFileWriter Class

Daniel Campbell
School of Engineering and Computer Science
Victoria University of Wellington, New Zealand
daniel.campbell@ecs.vuw.ac.nz

February 7, 2014

Contents

Chapter 1

Types

The Whiley programming language is *statically typed*, meaning that every expression has a type determined at compile time. By contrast, Javascript is dynamically typed, and will readily coerce variables of one type into another type. Because the default Javascript objects contain very little type information, it was necessary to create custom objects to represent Wyscript types in Javascript code - without these, nearly all type information would be lost once the code began running, making cast and instance-of checking impossible.

1.1 Primitive Types

Primitive values are the atomic building blocks of all data types in WyScript. Of all the primitive types, only the null, bool and void types are represented in Javascript with their corresponding primitive value - this is because these three types cannot be cast, and typechecking on them is trivial. Every other primitive type is represented as a specialised Javascript object, all of which are stored within the Wyscript object in the Wyscript.js library. These objects exist as *Expr.Constant*s in the AST, and are handled by the FileWriter when writing those expressions.

```
Primitive Value  ::=
                  |   Null    ->  null
                  |   Void    ->  (Cannot be instantiated)
                  |   Bool    ->  boolean
                  |   Int     ->  Wyscript.Integer
                  |   Real    ->  Wyscript.Float
                  |   Char    ->  Wyscript.Char
                  |   String  ->  Wyscript.String
```

1.1.1 Numbers

FileWriter. In the JavaScriptFileWriter, whenever a number literal is encountered (in the form of an `Expr.Constant`), it is wrapped in the corresponding Wyscript object - as all number literals are parsed into constants, this ensures that no native number literals are left in the converted file. In addition, any call to a binary expression (`Expr.Binary`) on the number is transformed into the appropriate method call.

```
int x = 1      →  var x = new Wyscript.Integer(1);
int y = 2      →  var y = new Wyscript.Integer(2);
int z = x - y  →  var z = new Wyscript.Integer(x.sub(y));
```

Library. The two number types are almost identical - they contain the *add()*, *sub()*, *mul()*, *div()* and *rem()* methods, which takes another number as a parameter and returns a new object containing the result (a real will always return a `Wyscript.Float`, an int will return either a `Wyscript.Integer` or a `Wyscript.Float` depending on the type of the parameter). These methods exist because there is no way to overload operators in Javascript. They also contain the equals and toString methods, and finally the `Wyscript.Integer` contains a cast method, which is used to promote it to a real (the only valid cast from one primitive type to another).

Notes. Note that all the methods of the number types return a new object, and do not alter the original object, ensuring Wyscript's pass by value semantics. Also, both objects have a type field, which is automatically assigned when the object is created. Also note that none of these methods, nor any other methods that return a number value, will return a native Javascript number - all numbers are wrapped in one of these two objects.

1.1.2 Text Values

FileWriter. Much like the number constants above, whenever a text literal is encountered (as an `Expr.Constant`) it is wrapped in the corresponding Wyscript object. In addition, any attempts to get the index'th character of the string (an `Expr.IndexOf`) are transformed into a *getValue(index)* call. Length expressions (`Expr.Unary`) are converted into a call to *length()*, and append expressions (`Expr.Binary`) are converted into a call to *append(other)*. Finally, when converting an assignment statement, a check is made for assigning a character of a string - this is transformed into an assignment with the value of a call to *assign(index, char)*.

```
string s = "Hello"  →  var s = new Wyscript.String("Hello");
s[0] = 'Z'          →  s = s.assign(0, 'Z');
```

Library. The `Wyscript.Char` object is very simple - it simply has a *toString()* method. The `Wyscript.String` object is more complex, as it shares a couple of methods with the `Wyscript.List` object - *getValue(index)* returns the index'th character of the string, *assign(index, char)* returns a new string with the index'th character replaced with char, *length()* returns the length of the string and *append(other)* returns the concatenation of the string with the string representation of other.

Notes. Note that all the methods of the text types return a new object, and do not alter the original object, ensuring Wyscript's pass by value semantics. Also, both objects have a type field, which is automatically assigned when the object is created. Finally, note that all the toString methods return an instance of `Wyscript.String`, with the exception of `Wyscript.String` itself, which returns a native string. To ensure you have a native string, call *toString()* twice - as Javascript native strings also have a (trivial) toString() method.

1.2 Composed Types

These are the objects that are composed from one or more other objects - due to their complexity they all have Javascript object representations. In addition, their type field is not determined statically, but created from an additional parameter passed to the constructor.

```
ComposedType  ::=
                 |   ReferenceType  ->  Wyscript.Ref
                 |   ListType       ->  Wyscript.List
                 |   RecordType     ->  Wyscript.Record
                 |   TupleType      ->  Wyscript.Tuple
```

1.2.1 References

FileWriter. References are handled in three places in the FileWriter - when handling an `Expr.New` (converted into creating a new `Wyscript.Ref`), when handling a dereference (converted into a call to *deref()*), and when handling a dereference assignment - this is transformed into a call to *setValue(value)*.

Library. A `Wyscript.Ref` consists of a value and a type - the type is an instance of `Wyscript.Type.Reference`, with the value's type as a parameter. in addition to the standard *toString* method, it has a *deref()* method, which returns its inner value, and the *setValue(newValue)* method, which reassigns the reference to a new value.

Notes. Note that reference values are not cloned and have no clone method - as reference values, they are the only objects which do not follow WyScript's pass-by-value semantics.

1.2.2 Lists

FileWriter. Lists are handled in several different places in the FileWriter, most of which are exactly the same as for strings. The only exception is that lists are initialised (and converted to their Wyscript equivalent), in a `Expr.ListConstructor`. Note this method also writes the type of the list, as it is required by the constructor.

Library. A `Wyscript.List` consists of an inner native array, and a type, which is passed as a parameter to the constructor (it will always be an instance of `Wyscript.Type.List`). In addition to the standard *toString()* and *equals(other)* methods, lists have a handful of other methods - they share *getValue(index)*, *setValue(index, value)* and *length()* with the `Wyscript.String` type, and it also has a *clone()* method, which performs a deep clone of the object (to ensure pass-by-value is upheld).

Notes. Many constructors take an array as a parameter (lists, records, tuples and their associated type objects) - do not pass these a `Wyscript.List`, as they are expecting a native array (this was done to keep code as brief as possible, and also prevents an infinite loop when writing a list constructor).

1.2.3 Records

FileWriter. Record literals (`Expr.RecordConstructor`) are converted into their Wyscript equivalent when the FileWriter writes an expression of that type. The other cases in the FileWriter involving records are when writing a `Expr.RecordAccess`, which is converted into a call to *getValue(name)*, and the case where assigning a value to a record's field, which is converted into a call to *setValue(name, value)*.

```
{int x, int y} point = {x:1, y:2}  →  var point = new Wyscript.Record(
                                         ['x', 'y'],[1, 2],
                                         new Wyscript.Type.Record(
                                            ['x', 'y'],
                                            [new Wyscript.Type.Int(),
                                            new Wyscript.Type.Int()]));
point.x = 2                        →  point.setValue('x', 2);
```

Library. A `Wyscript.Record` consists of two arrays and a type. The first array is a list of the names of the fields in the Record, the second array is a list of the corresponding types for each field (These lists must be equal in size). The type is passed to the constructor, but is guaranteed to be an instance of `Wyscript.Type.Record`. In addition to the standard *toString()* and *equals(other)* methods, records have a handful of other methods - *getValue(name)* returns the value associated with a given field (or null), *hasKey(name)* returns whether or not the record has a field with the given name, and *setValue(name, value)* associates the given value to the given name (name must be an existing field of the record). Finally, like the list and tuple types, it has the *clone()* method, which performs a deep clone of the object (to ensure pass-by-value is upheld).

Notes. The order of names passed into a record does not matter (so long as the types passed in have their order changed accordingly) - when printed a record sorts its field names, so that a record with the same effective type will always output the same way, regardless of how that type was declared.

1.2.4 Tuples

FileWriter. Tuples in Wyscript can be thought of as records with anonymous fields - however, unlike records they do not have setter/getter methods, as unlike records, the internal values of a tuple cannot be changed once instantiated. The only places where the FileWriter deals with tuples is in the `Expr.Tuple`, where tuples are created (this is converted into a new *Wyscript.Tuple*), and in assignment, where the values in a tuple are decomposed into some variables - in this case, the FileWriter stores the tuple in a temporary variable, and creates an assignment statement for each variable on the lhs.

```
(int, int) point = (0, 1)  →  var point = new Wyscript.Tuple([0, 1],
                                 new Wyscript.Type.Tuple([int, int]));
int x                      →  var x;
int y                      →  var y;
(x, y) = point             →  var WyscriptTupleVal = point;
                               x =  WyscriptTupleVal.values[0];
                               y =  WyscriptTupleVal.values[1];
```

Library. A `Wyscript.Tuple` consists of an array and a type. The array is simply the list of all the values in the tuple. The type is passed to the constructor, but is guaranteed to be an instance of `Wyscript.Type.Tuple`. Tuples have no specialised methods - they only have the *toString()*, *equals(other)*, and *clone()* methods. The values in a tuple object are instead accessed by accessing the tuple's inner *values* field.

1.3 Type Objects

In addition to the objects representing Wyscript data values, there exists Javascript object representations of the types themselves as well - these are mainly used for casting, and for checking instance-of with the *is* operator. They are written with the FileWriter's *write(Type t)* method, which operates similarly to the other *write* methods.

```
Type  ::=
       |   Null       ->  Wyscript.Type.Null
       |   Void       ->  Wyscript.Type.Void
       |   Bool       ->  Wyscript.Type.Bool
       |   Int        ->  Wyscript.Type.Int
       |   Real       ->  Wyscript.Type.Real
       |   Char       ->  Wyscript.Type.Char
       |   String     ->  Wyscript.Type.String
       |   Reference  ->  Wyscript.Type.Reference
       |   List       ->  Wyscript.Type.List
       |   Record     ->  Wyscript.Type.Record
       |   Tuple      ->  Wyscript.Type.Tuple
```

Chapter 2

Functions

In addition to encoding objects to represent Wyscript's type system, the Wyscript.js library also contains a handful of functions - these are generally functions that encode an expression that was too complex to be translated on a single line, such as a type casting operation.

2.1 Binary Expressions

These operations were moved into a function, and not made into a method of one of the type objects, either because they operate on too many possible types (equality, and to a lesser extend less than/-greater than) and so would result in too much duplicated code, or because the function in question made more sense as a standalone method (the range function).

2.1.1 Range

The *range* function is called with the `..` operator. It takes two parameters for its lower and upper bounds, both integers (these can be Wyscript or native ints, though in practice always Wyscript integers). It returns a Wyscript list of size (upper-lower), filled with the integers from lower (inclusive) to upper (exclusive), with type `[int]`.

```
[int] range = 0..10  →  var range = Wyscript.range(
                             new Wyscript.Integer(0),
                             new Wyscript.Integer(10));
```

2.1.2 Equality

Wyscript encodes functions for three different equality methods (*gt*, *lt* and *equals*), which all take three parameters - the lhs, the rhs, and a boolean *isEqual* flag, which is used to encode whether or the method will return true if the values are equal. Besides that, the methods are straightforward - the largest of the three methods is the equals method, as it needs to convert any of the primitive types into their native equivalents before checking for (in)equality (the compound types all have an equals method defined).

```
Logical Operator  ::=
                       |   x <  y  ->  Wyscript.lt(x, y, false)
                       |   x <= y  ->  Wyscript.lt(x, y, true)
                       |   x >  y  ->  Wyscript.gt(x, y, false)
                       |   x >= y  ->  Wyscript.gt(x, y, true)
                       |   x != y  ->  Wyscript.equals(x, y, false)
                       |   x == y  ->  Wyscript.equals(x, y, true)
```

2.2 Typechecking and Casting

Although relatively straightforward in Wyscript, these operations are quite complex in javascript, as they require either the analysis or the alteration (possibly recursive) of type information.

2.2.1 Typechecking

All typechecking operations (`Expr.Is`) are turned into a call to *Wyscript.Is(object, type)*. This method first checks if type is one of the primitive types, and if so, returns whether or not object is an instance of the associated type. If type is a compound tpe, it calls *object.type.subType(type)*. This method, which all Type objects have, exactly emulates the subtyping rules present in the Wyscript compiler's Typechecker. The one exception is if type is a union type, as in this case there is no guarantee that object will have a **`type`** field (it may be a native javascript data type). In this case it first calls *Wyscript.getType(object)*, which returns the type of any object.

2.2.2 Casting

All typecasting operations (`Expr.Cast`) are turned into a call to *Wyscript.cast(object, type)*. As these casts have already been checked by the typechecker, they are guaranteed to be valid - currently the only valid non-trivial cast is one where an int is casted to a real (either as a primitive or part of a compound type), or where a union type is casted to one of it's sub-types (though casts of that type have no effect on Javascript code and so are simply omitted).

The cast method first checks the object being casted, and the type it is being casted to. If it is a primitive type, it returns the object, unless it is a cast from an int to a real, in which case it returns *object.cast()*. It also checks for the trivial case where the object is already the given type - in that case it returns object (or *object.clone()* if available).

Next it checks for the compound types (note that attempting to cast a reference is a syntax error). If it is a list, it first creates a new array. It then calls *Wyscript.cast* for every element of the list, passing in that element and the element type of the original cast list type, placing the resulting elements in the new array. Finally, it creates and returns a new `Wyscript.List` using the new array and the cast type.

If the element is a tuple, it again creates a new array, stepping through and calling *Wyscript.cast* on the elements of the original tuple - the only diffference being each element is casted to the appropriate element type from the new tuple type's typelist. Once done it creates and returns a new tuple.

The case for casting records is very similar, with the exception that a new names array is created (identical to the old names array) to properly clone the casted record - other than that, the process of stepping through the values array and casting to the associated new type is identical.

Chapter 3

Control Flow Statements

The most difficult language features to convert from Wyscript to Javascript were control flow statements that did not operate the same in the two languages - namely, switch statements and for-each loops, in part because the entire conversion process had to happen in the FileWriter - there was no way to use the Wyscript.js library to ease the conversion.

3.1 Switch Statements

In Wyscript, switches have explicit fallthrough with the `next` command. In addition, the switch expression can be more than just an integer value - it can also be a real, string, or list. However, Javascript switches follow C switches, with implicit fall-through, no equivalent to the next statement (a break statement which has no equivalent in Wyscript), and only able to have an enumerable type as the switch expression. For these reasons, it was not feasible to represent a Wyscript switch statement as a Javascript switch statement.

Instead, switches are converted into a long if-else chain, held inside a labelled while(true) loop. The label is always *$label*, with the value of the FileWriter's *switchCount* variable appended (*switchCount* is incremented whenever a switch statement starts being written, and decremented when the statement has been written). Just before the while loop, a variable (*$WySwitch*, with *switchCount* appended) is declared. This variable is used to hold the value the if-else conditions evaluate.

Each case statement (and the optional default statement) are converted into an if condition - if it isn't the first statement being evaluated, it is an else-if condition. The default statement is always written last as the final else, and if no default is given, an empty else is created (to ensure that a next in the final case of a switch properly exits the loop). At the end of every if/else body, a call to break the outer loop is appended, simulating the explicit fallthrough. In addition, when writing a case/default, the value of the next case expression (or null if none or the next is a default) is passed to the FileWriter's *write* methods. If a next statement is written, the current *$WySwitch* variable is set to have the value of that expression (or a random default value if the expression is null), and a call to continue the outer-loop is made, simulating fallthrough.

```
switch(x):
    case (0):
        print x
        next
    case (1):
        print x+1

BECOMES:

var WySwitch0 = x;
label0: while(true) {
    if(Wyscript.equals(WySwitch0, new Wyscript.Integer(0))) {
        sysout.print(x);
        WySwitch0 = new Wyscript.Integer(1);
        continue label0;
        break label0;
    }
    else if (Wyscript.equals(WySwitch0, new Wyscript.Integer(1))) {
        sysout.print(x.add(new Wyscript.Integer(1)));
        break label0;
    }
    else {
        break label0;
    }
}
```

(Note the $ have been omitted for formatting reasons)

3.2 For-Each Loops

In Wyscript, all for-loops have the form:

```
for i in list:
```

Where i is an index into the given list, and the loop iterates once for every element in the list. Javascript only has the classical for loop:

```
for (i = 0; i < 3; i++) {
```

As a result, it is necessary to translate a Wyscript for-each loop into a Javascript for loop. (Javascript does have a for-each loop, but it iterates over the enumerable properties of the object - not the desired behaviour).

Whenever the FileWriter begins writing a for-each loop, it first creates an empty object to hold all the temporary variables. The object is called *$WyTmp*, with the value of *forCount* appended. (*forCount* serves the same purpose as *switchCount*, and is incremented/decremented in the same way). That object then has two properties added - *list*, which holds the list being iterated over, and *count*, which holds the current index into the list. Then the for-each loop is rewritten as a classical for loop iterating from 0 to the size of the list. The loop body is written as normal, but an additional line is inserted in the beginning, initialising the index value to be the value at list[count]. This ensures any reference to the index variable of the original for-each loop are still valid.

```
for i in list1:
        print i

BECOMES:

var WyTmp0 = {}
WyTmp0.list = list1.clone();
WyTmp0.count = 0;
for (WyTmp0.count = 0; WyTmp0.count < WyTmp0.list.length();
        WyTmp0.count++) {
    var i = WyTmp0.list.getValue(WyTmp0.count);
    sysout.print(i);
}
```

(Note that the list is a `Wyscript.List`, not a native javascript array. Also, $ have been omitted for formatting reasons).