RhinoWithContinuations

Transcript of an explanation of Christopher Oliver on cocoon-dev in July 2002.

Flow scripts in Cocoon are written in JavaScript. Why JavaScript ? Because it's a high-level scripting language that looks a lot like Java, and also because we have a special version of the Rhino JavaScript interpreter that has the ability to "capture" the current execution state of a program.

This special version is hosted on Cocoondev.org.

The `org.mozilla.javascript.continuations` package introduces an additional interpreted mode for Rhino that supports tail-call elimination and first-class continuations. Currently this mode is selected by setting the optimization level of the current context to -2. It may also be selected by passing "-opt -2" on the command line to the Rhino shell or debugger, for example like this:

(shell🙂

```
% java -cp js.jar org.mozilla.javascript.tools.shell.Main -opt -2 file.js
```

(debugger🙂

```
% java -cp js.jar org.mozilla.javascript.tools.debugger.Main -opt -2 file.js
```

Features

Tail-call elimination

You might think the following code is faulty since it apparently would overflow the call-stack given a large enough value for 'limit':

```
function g(count, limit) {
   if (count == limit) {
      return "done";
   }
   return g(count + 1, limit);
}
```

In fact, with the "continuations" mode of Rhino this is not the case. The interpreter detects that the recursive call to 'g()' is in so-called "tail position" (meaning that there is no further code to execute after the call) and simply overwrites the current call frame with the new call to 'g()'. Thus no additional stack space is used in such cases.

Continuations

Rhino now supports first-class continuations. In Rhino a continuation is a JavaScript object that represents a snapshot of the state of an executing Rhino script – i.e the current call-stack – including each call-frame's program counter and local variables.

NOTE: *Captured continuation does not save the state of the variables, but rather pointers to the variables. Example: two continuations will share one instance of the local variable declared in the function, if those two continuations were created during execution of the same call of this function. – Vadim*

The term "Continuation" (borrowed from Scheme) is used because it represents the rest of, or the "continuation" of, a program. Each time you call a function, there is an implicit continuation – the place where the function should return to. A JavaScript Continuation object provides a way to bind that implicit continuation to a name and keep hold of it. If you don't do anything with the named continuation, the program will eventually invoke it anyway when the function returns and passes control to its caller. Now that it's named, however, you could invoke the continuation earlier than normal, or you could invoke it later (after it has already been invoked by the normal control flow). In the early case, the effect is a non-local exit. In the later case, it's more like returning from the same function more than once.

You can capture the continuation of a script by simply calling

```
new Continuation()
```

for example:

```
function someFunction(a, b) {
    var kont = new Continuation();
}
```

The variable `kont` now represents the execution state of the current caller of `someFunction`. (For those of you who are familiar with Scheme's `call-with-current-continuation`, here is the Rhino equivalent:

```
function call_with_current_continuation(fun) {
    var kont = new Continuation();
    return fun(kont);
}
```

Since `kont` is a first-class JavaScript object you can return it, store it in a variable, assign it to a property of another object - whatever you like. In addition, a Continuation object is also a function, which may be called. When you make such a call the current execution state of the program is discarded and the snapshot of the program represented by the Continuation object is resumed in its place. You may also pass an argument to the call to a Continuation object (if you don't pass an argument 'undefined' is implicitly passed instead). The value you pass as an argument to the Continuation object becomes the return value of the function in which the Continuation was captured. For example:

```
01  function someFunction()  {
02     var kont  = new  Continuation();
03     print("captured: " + kont);
04     return kont;
05  }
06
07  var k = someFunction();
08  if (k instanceof Continuation) {
09     print("k is a continuation");
10     k(200);
11  } else {
12     print("k is now a " + typeof(k));
13  }
14  print(k);
```

Evaluating the above script yields the following output:

```
captured: [object Continuation]
k is a continuation
k is now a number
200
```

When the continuation `k` is invoked on line 10, the program "jumps" back to the call to `someFunction` on line 7, but this time `someFunction` returns with the value '200' (which was passed into the call to `k` on line 10).

In addition, a Continuation object may be called more than once. Each time it is called it restarts execution at the return point of the function in which it was captured. This means that the same function invocation can return multiple times (and with different return values).

Finally, note that a Continuation created in a top-level script provides a means to terminate any script immediately. Whenever such a Continuation is invoked it simply terminates the interpreter, for example:

```
var suicide = new Continuation();

function foo(suicide) {
    print("commiting suicide");
    suicide();
    print("never reached");
}

foo(suicide);
```

ContinuationException

A Continuation can be thought of as representing the return from a function invocation. In JavaScript, in addition to a normal return, a function invocation may return due to an exception. In continuations mode, Rhino provides a special built-in object `ContinuationException` which allows you to throw an exception to a Continuation. ContinuationException's constructor takes one argument - the object you want to throw. When an instance of

ContinuationException is passed to a Continuation, the value of that argument

is thrown in the context of the Continuation after the Continuation is restored. For example:

```
function someFunction() {
    var k = new Continuation();
    return k;
}

try {
    var k = someFunction();
    print("k: " + k);
    if (k instanceof Continuation) {
        print("k is a continuation");
        k(new ContinuationException("this is thrown from someFunction"));
    }
    print("never reached");
} catch (e) {
    print("caught exception: " + e);
}
```

Evaluating the above script yields the following output:

```
k: [object Continuation]
k is a continuation
caught exception: this is thrown from someFunction
```

Controlling what gets captured in a Continuation

In continuations mode, Rhino provides a special extended syntax to allow you to control what gets captured in a continuation as follows:

```
catch (break) {
    // a continuation has been captured - code to handle that
    // goes here
}

catch (continue) {
    // a continuation has been resumed - code to handle that
    // goes here
}
```

Multiple such "catch" clauses may be present at any scope. All such clauses contained within a script or function invocation captured or resumed as part of a Continuation will be executed when that Continuation is captured or resumed. For example, you might want to return a pooled JDBC connection to its connection pool while a Continuation is suspended and recover the connection when the Continuation is resumed:

```
var pool = ...;

function someFunction() {

    var conn = pool.getConnection();
    ...

    catch (break) {
        conn.close();
        conn = null;
    }

    catch (continue) {
        conn = pool.getConnection();
    }
}
```

If you want to execute Flowscript code after calling the view layer but before control leaves the interpreter, catch(return) will help:

```
catch (return) {
    // after calling the view layer but before control
    // leaves the interpreter
}
```

See also http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=105825410721036&w=2

Continuations are Serializable

This version of Rhino also contains experimental support for serializing Continuations. Thus you may save the state of an executing script and restore it later. Here is an example:

```
function capture(filename) {
    var k = new Continuation();
    serialize(k, filename);
    java.lang.System.exit(0);
}

function foo(level) {
    var now = new java.util.Date();
    if(level > 5) {
        print("run the file foo.ser");
        capture("foo.ser");
    } else {
        print("next level");
        foo(level + 1);
    }
    print("restarted("+level+"): " + now)
}

foo(1);
```

Evaluating the above script saves a Continuation to the file "foo.ser" and prints the following output:

```
next level
next level
next level
next level
next level
run the file foo.ser
```

The following script deserializes the Continuation and executes it:

```
var k = deserialize("foo.ser");
k();
```

Evaluating the second script produces the following output:

```
restarted(6): Mon May 20 19:03:12 PDT 2002
restarted(5): Mon May 20 19:03:12 PDT 2002
restarted(4): Mon May 20 19:03:12 PDT 2002
restarted(3): Mon May 20 19:03:12 PDT 2002
restarted(2): Mon May 20 19:03:12 PDT 2002
restarted(1): Mon May 20 19:03:12 PDT 2002
```