Lambda-based control abstractions using break and continue
Allen Wirfs-Brock
March 15, 2012

The is growing interest in adding support to ECMAScript for a construct that is sometimes called a "block lambda". A block lambda is the definition of functions that adhere to Tennent's Correspondence principle which can be loosely paraphrased as:

Wrapping a block of code in a function that is immediate called should produce the same effect as directly executing the original bock of code.

Various syntactic forms have been proposed for expressing block lambdas in ECMAScript. The pros and cons of the various alternative syntaxes for block lambdas is not the concern of this note. In order to remain neutral with regard to the syntax debate, this note uses a hypothetical syntax that is not under considerations. Within this note, block lambdas are written using this syntactic production:

BlockLambda : λ(*FormalParameterList* $_{\text{opt}}$) { *StatementList* $_{\text{opt}}$ }

Using appropriately specified block lambdas and TCP we can take a function like this:

```
function foo() {
   for (let v of a) {
      if (v == value1) continue;
      if (v == value2) break;
      if (v == value3) return;
      doSomethingWith(v);
   }
 };
```

and convert it to something like:

```
function foo() {
   for (let v of a) λ(){
      if (v == value1) continue;
      if (v == value2) break;
      if (v == value3) return;
      doSomethingWith(v);
   }()  //← immediate call of block lambda
 };
```

and everything will continue to work exactly the same. In particular, in this situation the **continue**, **break**, and **return** statements will behave exactly the same whether or not they are enclosed in the block lambda.

However, what if we want to replace the for-of loop in the above example with use of the Array forEach method like this:

```
function foo() {
   a.forEach(λ(v) {
      if (v == value1) continue;
      if (v == value2) break;
      if (v == value3) return;
      doSomethingWith(v);
   });
 }
```

As block lambdas have so far been proposed, this would not work. The **return** statement is not a problem, as it has been defined to always return from the closest enclosing non block lambda function. In this case that is the function foo.

However, the **break** and **continue** statements are problematic. Using the semantics of **break** and **continue** defined by ES5.1, this version of the code would produce an early syntax error because a **continue** statement is only allowed to occur nested within the body of an *IterationStatement* and a **break** statement (without a target label) is only allow to occur nested within the body of an *IterationStatement* or a *SwitchStatement*.

One of the primary motivations for adding block lambda to ES is to provide a facilities that allows ES programmer to define control abstraction that have equivalent power to the languages built-in control statements. Block lambdas can supply the bodies of such control abstraction when they are passed as arguments to the functions that represent the abstraction. But, if we only have block lambdas as defined thus far, such control abstractions would still be second class citizens in comparison to the built-in control statements because the **continue** and **break** statements could not be used to control iteration.

In ES5.1 iteration statements (by which we really mean *IterationStatement* or *SwitchStatement*) each define specific semantics for what to do when a **continue** or **break** occurs within the body of the statement. The specific semantics actually varies slightly depending upon which specific kind of iteration statement is involved. The semantics are defined in the ES5.1 specification using the concept of Completion values. Completion values propagate from a control transfer point (a **return**, **throw**, **break**, or **continue** statement) to a dynamically (and often lexically) enclosing context that intercepts the control transfer. Completion values identify the type of transfer (return, break, etc.). Completion values essentially allow iteration statements to define the behavior of **continue** and **break** statements that occur within their bodies. For user defined looping abstractions to have comparable behavior they will also have to be able to define their own semantics for any **continue** or **break** that occurs within their block lambda provided "bodies".

However not all control abstractions need or want to control the handling of **continue** and **break**. Consider that somebody might have a reason to construct a control abstraction function that was equivalent to an if statement:

```
function ifElse(predicate, thenClause, elseClause) {
  if (predicate) return thenClause();
  else return elseClause();
};
```

If they also have a function such as this:

```
function bar() {
   for (let v of a) {
      if (v == value1) continue;
      else break;
   }
};
```

they might choose to refactor it to use their ifElse function:

```
function bar() {
   for (let v of a) {
      ifElse((v == value1), λ() {continue},
        λ(){break})
};
```

In this case, interception of the break and continue by the ifElse function would clearly change the meaning of the program and the coder of ifElse would want to be sure that control/break interception did not occur. Whether or not a control abstraction handles breaks and continues needs to be part of the definition of the abstraction, so let's see how we can enable the implementers to make that decision.

Consider a implementation of Array forEach as it might be defined in ES5.1:

```
Array.prototype.forEach = function(callBack, thisArg) {
   var index = 0;
   var value = undefined;
   while (index < this.length) {
      value = callBack.call(thisArg,this[index],index++,this);
   }
   return value;  //return value from last iteration.
}
```

A function like this is unable to manage the occurrence of a **break** or **continue** that is executed during an invocation of the callback function because it has no visibility of the occurrence. In order to fully emulate a built-in iteration statement such a function needs to be able to detect the occurrence of a label-less **break** or **continue** and needs to be able to specify what action to take upon such an occurrence. There are three plausible ways this might be accomplished:

1. Label-less **break** and **continue** statements could be respecified to throw an exception and the control abstraction function could catch such exception using a try-catch statement:

```
Array.prototype.forEach = function(callBack, thisArg) {
   var index = 0;
   var value = undefined;
   while (index < this.length) {
      try {
         value = callBack.call(thisArg,this[index],index++,this);
      } catch (e) {
        if (e.name == "break") break;  //from while loop
        if (e.name == "continue") continue //the while loop
        throw e;    //rethrow unexpected exception
   }
   return value;  //return value from last iteration.
}
```

2. The callBack function could be invoked in a special manner that allows the caller to parameterize what action is taken if an label-less **break** or **continue** occurs within the function. For the following example, assume that callCtl is a method of functions that is just like the call method except that it takes two additional leading arguments which are functions that deal with break and continue conditions, respectively.

```
Array.prototype.forEach = function(callBack, thisArg) {
   var index = 0;
   var value = undefined;
   while (index < this.length) {
      value = callBack.callCtl(
         λ(){break}, //from while loop, if break occurs in callBack
         λ(){continue}, //the while loop, if continue in callBack
         thisArg, this[index], index++, this);
   }
   return value;  //return value from last iteration.
}
```

3. The callBack function could be invoked in a special manner that returns a reified Completion value. The caller could check the completion value for to see if a **break** or **continue** occurred within the function and respond appropriately. For the following example, assume that callCV is a method of functions that is just like the call method except that it always returns an object that represents a Completion value object.

```
Array.prototype.forEach = fn(callBack, thisArg) {
   var index = 0;
   var completion = {value: undefined];
   loop: while (index < this.length) {
      completion = callBack.callCV(thisArg,this[index],index++,this);
      switch (completion.kind) {
         case "break":
             break loop;  //out of implementation while loop
         case "continue":
         default:
            continue loop;  //next iteration, not really needed here
       }
    return completion.value; //return value from last itr completion
```

```
}
```

Of these three approaches, the exception based scheme seems to be the most potentially problematic and error prone for the ES programmer. Because ES exception handling is untyped, exception handling for control flow may need to be intermingled with exception handling for other purpose and break/continue exception can be easily missed or inadvertently caught introducing hard to find bugs.

The other two approaches are essentially duals of each other that operate upon opposite sides of the [[Call]] interface. Recall that the ES5.1 [[Call]] internal method, after evaluating a function, takes the resulting completion value produced by the code and for normal completions converts them to simple values that returned to the caller while abrupt completions are propagated as exceptions. The callCtl approach essentially parameterizes the [[Call]] method with explicit actions to take for break and continue abrupt completions. The callCV approach takes the responsibility of interpreting certain abrupt completions away from the [[Call]] method and instead reifies the Completion value as an ECMAScript object that is returned to the original called. It is then the caller's responsibility to decide what to do with **break** and **continue** completions.

Of the latter two approaches, it isn't immediately clear that one is obviously preferable. In the above examples, the callCtl approach is more compact and perhaps easiest to read. However, in the callCV approach a simple one line **if** statement could replace the **switch** statement in this instance and a simpler version would look like this:

```
Array.prototype.forEach = fn(callBack, thisArg) {
   var index = 0;
   var completion = {value: undefined];
   while (index < this.length) {
      completion = callBack.callCV(thisArg,this[index],index++,this);
      if (completion.kind == "break") break;
   }
   return completion.value; //return value from last iteration
}
```

At least for this abstraction, the simplified callCV version probably beats the callCtl approach on both conciseness and clarity.

There may also be performance differences between the approaches. The callCtl approach requires the creation of two block lambdas for every callCtl that is executed (but perhaps an optimizer could treat them as loop invariant values) and cross frame break/continue escapes. The callCV approach requires creation of the Completion object but only has normal LIFO returns. It seems plausible, that the callCV approach can have less runtime overhead using simpler optimization techniques.

Other examples (these aren't complete or fully worked out)

```
Collection.prototype.forEachAlternating = fn(callBack1, callBack2) {
   let first = true;
   let value = undefined;
   return this.forEach(λ(v) {
      value = (first?callBack1:callBack2).callCtl(
         λ() {break},  //break from forEach
         null,  //do nothing for continue, just returns
         this, v);
      first = !first; //next iteration will use other callBack
      value;
   });
 }
```

```
Collection.prototype.forEachAlternating = fn(callBack1, callBack2) {
   let first = true;
   let completion = {value: undefined};
   return this.forEach(λ(v) {
      completion = (first?callBack1:callBack2).callCV(this, v);
      if (completion.kind is "break") break; //from forEacj
      first = !this; //next iteration will use other callback
      completion.value;
    });
}
```

```
Collection.prototype.cascade = fn(firstThis,...args) {
   let value = undefined;
   let lastValue = firstThis;
   return this.forEach(λ(f) {
      lastValue = f.applyCtl(
         λ() {break},  //break from forEach
         null,  //do nothing for continue, just returns
         lastValue, args);
      first = !first; //next iteration will use other callBack
      lastValue;
   });
 }
```

```
FSM(
  0, λ() {c = nextChar(); n=0; continue with 1},
  1, λ() {if (isDigit(c)) {
              n=10*n+code(c)-code('0');
              c = nextChar();
              continue with 1};
```

```
          else continue with 2},
  2, λ() {if (isWhiteSpace(c)) {




c = nextChar(); continue with 1},
[λ() { },
 λ() {if (this  },
 λ() { }].do(2);
```