Ecma/TC39/2010/034

Shallow continuations

Syntax

```
UnaryExpression ::= ... | "shift" UnaryExpression
```

Semantics

Every function containing the `shift` operator in its body (not contained in any nested functions) implicitly creates a *continuation object* every time it is called. The continuation object encapsulates the function call's activation object.

Let the current stack consist of *S* followed by activation object *A*. Let *k* be the continuation object of *A*. Let *o* be the binding of `this` in *A*.

1. Evaluate the argument expression to get a value *v*.
2. Capture the activation object *A* and set the internal `[[Value]]` property of `k` to *A*.
3. Remove *A* from the stack, leaving *S* as the current stack.
4. Call *v* with *k* as its single argument, with `this` bound to *o*.

Continuation objects

A continuation object *k* has the following methods:

- `send(x)`:
  1. Let *A* be the activation object in *k*'s `[[Value]]` property.
  2. Push *A* onto the current stack.
  3. Use the value of `x` as the result of the suspended `shift` expression and continue evaluating the function activation.
- `throw(x)`:
  1. Let *A* be the activation object in *k*'s `[[Value]]` property.
  2. Push *A* onto the current stack.
  3. Use the value of `x` as an exception value to throw in place of the suspended `shift` expression and continue evaluating the function activation.
- `close()`:
  1. Let *A* be the activation object in *k*'s `[[Value]]` property.
  2. Push *A* onto the current stack.
  3. Replace the suspended `shift` expression with `let () { return }`.

Note that suspending the same activation object multiple times leads to the same (in the sense of ===) continuation object.

Statefulness

Note that every subsequent evaluation of `shift` for the same activation *A* produces the same continuation object *k*. This means that as execution of *A* proceeds, the continuation object *k* reflects the changing state of the computation.

Examples

```
function f() {
    try {
        for (let i = 0; i < K; i++) {
            farble(i);

            // suspend and return the activation
            let received = shift function(k) { return k };

            print(received); // this will be 42
        }
    }
    catch (e) { /* ... */ }
}

let k = f(); // starts running and return the suspended activation
// ...
k.send(42); // resume suspended activation
```

Enhancements

- allow the argument to be optional, defaulting to `function (k) { return k }`

Generators

Generators are a convenience form for creating custom iterators.

Syntax

```
PrimaryExpression ::= ... | "generator" Identifier? FunctionArguments FunctionBody

GeneratorDeclaration ::= ... | "generator" Identifier FunctionArguments FunctionBody
```

Within a generator body, it is a syntax error for `return` to take an argument expression.

Semantics

The expression `generator(x1,...,xn) { body ... }` is equivalent to:

```
function(x1,...,xn) {
    shift function(k) {
        return Object.freeze({
            send:  function(x) { return k.send(x); },
            next:  function()  { return k.send(void 0); },
            throw: function(x) { return k.throw(x); },
            close: function()  { k.close(); }
        });
    };
    body ...
    throw StopIteration;
}
```

for the original definitions of `StopIteration` and `Object.freeze`.

Inside a function body, a `return` statement (which may not have an argument) is equivalent to:

```
throw StopIteration;
```

for the original definition of `StopIteration`.

The expression `yield e` is equivalent to:

```
let (result = e) { => (shift function(k) { return result; }) }
```

A couple of quick notes before the TC39 meeting:

- `yield` is a low-precedence operator, at the same precedence as assignment in Python and JS1.7+. This means in an argument list or comma expression, you must parenthesize on the outside: `foo(a, (yield b), c)`. Python requires parenthesization even if there's only one argument: `foo`[1], but JS1.7+ do not.
- `generator` instead of `function` (in JS1.7+, after Python which reuses its `def` function-declaring keyword) has benefit in terms of explicitness (you don't have to look for `yield` usage in the body to know it's a generator, not a function), but breaks the ability to use `const` instead of `function` (see const functions). `const generator` is a bit much, and not parallel (no `const function`).

— Brendan Eich 2010/05/24 04:47

---

[1] yield bar