Ecma/TC39/2010/039

Const Functions

This page proposes to allow the `const` keyword to appear wherever the `function` keyword is allowed, with the following consequences:

- The defined function is born frozen.
- The defined function's `prototype` property is born frozen.
- If the form defines a named variable, then the variable is unassignable, as if defined as a `const` variable.
- If the form is a named function declaration, then, like the normal named function declaration, the named variable is initialized at the beginning of its block, rather than where the declaration appears.

ExpressionStatement

```
ExpressionStatement:
    [lookahead not-in { "{", "function", "const" }] Expression ";"
```

Just as an ExpressionStatement cannot begin with `function`, it would also not be able to begin with `const`.

FunctionDeclaration

```
FunctionDeclaration:
    "function" Identifier "(" FormalParameterList? ")" "{" FunctionBody "}"
    "const" Identifier "(" FormalParameterList? ")" "{" FunctionBody "}"
```

For example, the semantics of the FunctionDeclaration `const foo(a) {return a(foo);}` is equivalent to

```
const foo = function(a) {return a(foo);};
Object.freeze(foo);
Object.freeze(foo.prototype);
```

where those three lines are hoisted to the top of the block containing this function definition, so that

- the variable `foo` cannot be observed in a non-initialized state (and so no read barrier is needed).
- the function cannot be observed in a non-frozen state.
- the value of the function's `prototype` property cannot be observed in a non-frozen state.

FunctionExpression

```
FunctionExpression:
    "function" Identifier? "(" FormalParameterList? ")" "{" FunctionBody "}"
    "const" Identifier? "(" FormalParameterList? ")" "{" FunctionBody "}"
```

The semantics of the FunctionExpression `const foo(a) {return a(foo);}` is equivalent to the expression

```
(function(){
  const foo = function(a) {return a(foo);};
  Object.freeze(foo);
  Object.freeze(foo.prototype);
  return foo;
})()
```

The expansion here is a bit trickier. However, since two function boundaries do not violate TCP (tennent correspondence principle) any more than one, this works.

The semantics of the FunctionExpression `const(a) {return a;}` is equivalent to the expression

```
(function(func) {
  Object.freeze(func);
  Object.freeze(func.prototype);
  return func;
})(function(a) {return a;})
```

This expansion carefully avoids any conflict with other possible uses of the parameter name `func`. (Obviously, a hygienic expansion system can avoid such name conflicts without resort to such games.)

Examples

High Integrity Factories

Const functions combined with ES5's `Object.freeze` provide a more convenient syntax for high integrity factories than anything that can be expressed in ES5 by itself.

```
const Point(x, y) {
  return Object.freeze({
    getX: const() { return x; },
    getY: const() { return y; },
    add: const(otherPt) {
      return Point(x + otherPt.getX(),
                   y + otherPt.getY())
    },
    toString: const() { return '<' + x + ',' + y + '>'; }
  });
}
```

or, if we wish the factory to have the `instanceof` behavior associated with constructors, we can also make use of ES5's `Object.create`.

```
const Point(x, y) {
  return Object.freeze(Object.create(Point.prototype, {
    getX: {value: const() { return x; }},
    getY: {value: const() { return y; }},
    add: {value: const(otherPt) {
      return Point(x + otherPt.getX(),
                   y + otherPt.getY())
    }},
    toString: {value: const() { return '<' + x + ',' + y + '>'; }}
  }));
}
```

Of course, sweeter sugar such as Classes as Sugar would make high integrity factories even easier.

strawman/const_functions.txt · Last modified: 2010/07/01 17:59 by markm