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", "class" }] Expression ";"
```

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

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.

In all the expansions shown on this page, when we use `Object.freeze`, we actually mean the original binding of `Object.freeze`, not the current binding. In this sense, these expansions are as-if hygienic, and so are not simply naive syntactic sugar.

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.)

Joining

Often, convenient coding patterns will express more function evaluations than are really needed. For example:

```
function divisible(m) {
  return function(list) {
    return list.filter(function(n) {
      return n%m === 0;
    });
  };
}
const even = divisible(2);
```

The anonymous function passed to `filter` does not capture any variables defined by its immediately enclosing block, so it seems wasteful to evaluate it to a fresh closure with a fresh identity each time the expression is evaluated. The text of the EcmaScript 3 (ES3) spec sought to allow an optimization equivalent to the following rewrite (where variable names ending with triple underbar are assumed not to conflict with any other identifiers):

```
// BAD NON-TRANSPARENT JOINING OPTIMIZATION
function divisible(m) {
  function t1___(n) {
    return n%m === 0;
  }
  return function(list) {
    return list.filter(t1___);
  };
}
const even = divisible(2);
```

However, when these are full functions, that "optimization" is hardly transparent, since each of the original's generated functions is separately mutable and has a `.prototype` object which is also separately mutable. All major implementations of ES3 (and all implementations of which I'm aware) wisely avoided this "optimization" because of this problem; and the ES5 spec no longer allows this optimization. However, using const functions instead:

```
const divisible(m) {
  return const(list) {
    return list.filter(const(n) {
      return n%m === 0;
    });
  };
}
const even = divisible(2);
```

the corresponding joining optimization

```
const divisible(m) {
  const t1___(n) {
    return n%m === 0;
  }
  return const(list) {
    return list.filter(t1___);
  };
}
```

```
  const even = divisible(2);
```

clearly preserves the intent of the original but for the separate identities of the function and its prototype object. Thus, it makes sense to require this optimization. We propose the following fixpoint rule:

Loop

- For each `const` FunctionDeclaration or FunctionExpression *f*,
  - Let *s* be the outermost lexical scope (strict block, function, or program) enclosing *f*.
  - for each variable *v* used freely in *f*,
    - If *v* is defined by a declaration (`let`, `const`, `var`, or `function`) in an enclosing lexical scope *s1*,
      - Let *s* be the nearest of *s* and *s1*.
  - *s* is now the outermost scope at which *f* could have been declared.
  - Allocate *f* once per entry of *s*, as if it had become a declaration in *s* of an unmentioned variable.
- If no const functions were promoted by the above steps, exit.

This is a simple deterministic rule that is easy to implement and provides virtually all the benefit that a more unpredictable "implementation defined" exemption would allow. By this rule, the expansion shown above is a fixpoint, since the free `m` prevents `t1___` from being promoted further, and thus the free `t1___` in the remaining anonymous function prevents it from being promoted further.

Degenerate Case May Be Common

The example above limits hoisting in order to better explain the proposed mechanics. However, with the availability of array generics, the common case may be the degenerate case where a const function can be promoted all the way to top level, giving maximal benefit. Using this same example, it is now concise enough to extract the even members of a list where needed that one-off uses will often just open code it in place:

```
  //...
    //...arbitrarily nested...
    var evenList = list.filter(const(n) { return n%2 === 0; });
    //..
  //...
```

Since the const function above is closed (has no free variables), it would get promoted all the way.

```
const t1___(n) { return n%2 === 0; }
//...
  //...arbitrarily nested...
  var evenList = list.filter(t1___);
  //..
//...
```

By adopting this promotion rule, the program notation avoids the distraction cost of this non-local code organization while still painlessly obtaining all the benefits.

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: 2010/09/05 14:12 by markm

Show pagesource Login Index

Old revisions Back to top