Interpreting JavaScript

CS565

Purdue University

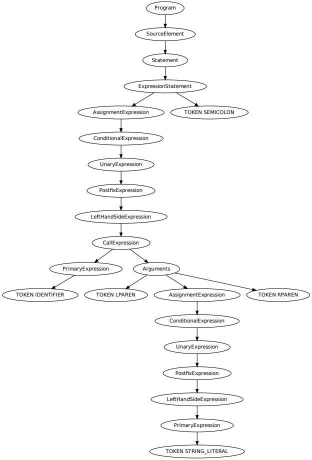
January 28, 2010

Methods of Interpretation

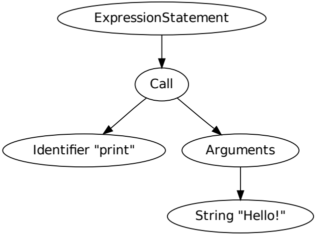
- **AST-walking**
- Bytecode
- JIT
- AOT (not actually interpretation, but sometimes made to look like it)

AST

Abstract Syntax Tree, internal tree representation of code



→



AST Design

- No one correct, canonical AST
- Design the AST however is convenient for you
- You will *not* be graded on the AST
- (Note: Most hand-written parsers produce an AST directly)

Interpretation — AST-walking

- Exactly what it says on the tin
- Recurse over the tree, performing each operation
- Simplest, but slowest, means of interpreting

AST-walking example

```
function interpretCall(context, callnode) {
    func := interpretNode(context, callnode.children[0])
    # JS uses eager evaluation
    args := interpretNode(context, callnode.children[1])

    funcbody := func.getFunctionBody()

    # create a new context in which the function body
    # will be evaluated, then ...
    return interpretNode(newcontext, funcbody)
}
```

AST-walking consequences

- Recursive calls: Slow!
- Store context in an object that's passed down through the interpret calls
- Context includes:
    - Local variables (scope)
    - Reference to outer scope
    - Current value of `this`
    - Anything else you may need

Types

(Not to be confused with prototypes)

- Undefined
- Null
- Boolean
- String
- Number
- Object

Types — To Object, or not to Object

Values have differing degrees of object-like behavior:

- Undefined, Null: No fields, no prototype, attempt to access fields throws exception
- Boolean, String, Number: No own fields, have a prototype, access fields through prototype
- Object: Own fields and prototype

"Specification types"

Types that are used in the specification, but are not generally part of implementation (used to explain semantics)

- Reference: Used to explain `delete`, `typeof`, assignment
- List: Used mainly for arguments
- Completion: Nonlocal control transfer
- Property descriptor, property identifier: Used for attributes of properties (we'll get to these in a few slides)

Primitive Types

All types but Object are primitive.
Number madness:

- Try not to scrutinize the section on Numbers too much
- `double`
- NaN: `NAN` in **math.h** or 0.0/0.0
- Infinity: `INFINITY` in **math.h** or 1.0/0.0

Type conversion

Implicit type conversions common:

```
var a = 2 - "1";
var b = (new Object()) + ",␣world!";
```

But how? (See ECMA-262 section 9)
For Objects:

```
Object x to String:
    if (x.toString is a method) return x.toString()
    if (x.valueOf is a method) return
                                toString(x.valueOf())
    throw TypeError
```

Properties

ECMA-262 8.6.1
Properties have these attributes:

- Value
- Writable
- Enumerable
- Configurable

Accessor properties have these attributes:

- Get
- Set
- Enumerable
- Configurable

Object internal properties/functions

ECMA-262 8.6.2. These are mostly used to document semantics

- Prototype
- Class (because object, prototypes and types weren't enough!)
- Extensible
- Get(propertyName) (value)
- GetOwnProperty(propertyName) (property descriptor)
- GetProperty(propertyName) (property descriptor)
- Put(propertyName, value, throwErr)
- CanPut(propertyName)
- HasProperty(propertyName)
- Delete(propertyName)
- DefaultValue(hint)
- DefineOwnProperty(propertyName, descriptor, throwErr)

More internal properties

ECMA-262 8.6.2. These are used for special types of objects

- PrimitiveValue
- Construct
- Call
- HasInstance (think instanceof)
- Scope
- FormalParameters
- Code
- TargetFunction
- BoundThis
- BoundArguments
- Match (for REs)
- ParameterMap (for `arguments`)

Classes

These describe what extra internal properties are provided

- Arguments
- Array
- Boolean
- Date
- Error
- Function
- Math
- Number
- Object
- Number
- Object
- RegExp
- String

Types or classes?

Notice that Boolean, Number, String are both types and classes
This reflects optional boxing

Object implementation

- Hashtable, prototype, primitive data (for numers etc), other properties described above
- Property lookup:
  - Look in hash table
  - If found, done
  - If not found
    - If prototype is null, fail
    - Otherwise, restart looking in prototype
- Feel free to *not* implement unboxed types (that is, only have an Object type)

Passing objects and non-objects

If you do pass things unboxed, it must be possible to distinguished objects from values

Simplest method:

```
struct JSValue {
    int type; /* references some enum */
    union value {
        struct JSObject *obj;
        double dbl;
        unicodestr *str;
        ...
    }
};
```

There are better ways

Functions

Some objects happen to be functions!

```
function interpretCall(context, func, this, args) {
    # Make a new context
    ncontext := new context with scope as func.scope and "this" set

    # Put together the arguments
    ncontex.locals.addArguments(func.parameters, args)
    arguments := new Arguments object from func.parameters, args (ECMA-262 10.6)
    ncontext.locals.set("arguments", arguments)

    # Perform the call
    result := interpretNode(ncontext, func.body)

    # And return the result
    return result
}
```

Functions — This

But where did this come from?

```
foo.devour(candy);
```

Call expression needs to know when a dot-expression is being called

```
devour(candy);
```

Otherwise, "this" is global object

Dynamic evaluation

Most code known at parse-time. Except not.

```
eval("print(\"Hello,␣evil!\");");
var x = new Object();
x.foo = "Hello,␣dynamically␣generated␣member␣names!";
print(x["f" + "oo"]);
```

The only lesson here is that you need to be able to parse late

Exceptions

```
  throw new ObnoxiousError();
```

Now what?

Exception implementation

Simplest and most portable way

- Handle exceptions like errors from C functions
- Return with a special "this is an exception" flag
- **Every** time you get a value, check if it's an exception first
- Slooooow, complicated, easy to screw up, but so easy (?)

Exceptions — setjmp/longjmp

A modicum less portable, far less simple, and much, much, much better

```
#include <setjmp.h>
...
jmp_buf env;
if (setjmp(env)) {
    /* catching an exception */
} else {
    ...
    /* throwing an exception */
    longjmp(env, 1);
}
```

Need to keep a stack of jmp_bufs in (you guessed it) the context

Another dive into the ECMA spec! Or, how to read ECMA