Lecture 5: Interpreter runtime structures

David Hovemeyer

September 13, 2023

601.428/628 Compilers and Interpreters



▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

► Scopes, environments, function calls

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

- Runtime data structures
- Reference counting

Scopes, environments, and function calls

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

Scope: in what region(s) of the program is a particular variable visible?

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … 釣�?

- ▶ Lifetime: *when* in the execution of the program does a variable exist?
- These are related but distinct concepts

```
var a, b, c;
a = 1;
b = 2;
c = 3;
function add1(a) {
  var b;
  b = 1;
  c = 4;
  a + b;
}
var d;
d = add1(c);
println(a);
println(b);
println(c);
d;
```



scope of global variable a





scope of global variable b



scope of global variable b

global variable & not visible here: shadowed by local variable

var a, b, c; a = 1; b = 2: c = 3:function add1(a) { var b; b = 1;c = 4;a + b;} var d; d = add1(c);println(a); println(b); println(c); d;

global variable c's scope is the entire program _not shadowed by any identically-named parameters or local variables



global variable c's scope is the entire program _not shadowed by any identically-named parameters or local variables

var a, b, c; a = 1; b = 2; c = 3;



scope of parameter a: body of function

var d; d = add1(c);

```
println(a);
println(b);
println(c);
d;
```

var a, b, c; a = 1; b = 2; c = 3;



scope of local variable b: from point of definition to end of function body

```
var d;
d = add1(c);
```

println(a);
println(b);
println(c);
d;

var a, b, c; a = 1; b = 2;c = 3;function add1(a) { var b; b = 1;c = 4;a + b;} var d d = add1(c);println(a); println(b); println(c); d;

- Global variables exist for the duration of the execution of the program
- Parameters and local variables exist for the duration of a function call
 - Call stack: each call pushes an activation record
 - A calls B, B calls C, C calls D, etc. arbitrarily many calls can be in progress at any point

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

- In practice, the call stack is usually limited in size
- Recursion: A calls itself
 - Caller and callee always have distinct activation records

- We'll use the term *environment* for a data structure containing a collection of variables that have a common lifetime
- Global environment: has definitions of global variables
 - Global variables are visible throughout the program unless shadowed by a variable in an "inner" scope
- Function call environment: created to represent parameters of a called function
- Block (statement list) environment: created to accommodate local variables defined in a block (statement list enclosed by curly braces)

- Nesting: an "inner" environment can reference variables in an "outer" environment
 - But not vice versa!
- ► In a *block-structured* language, every "block" defines a new environment
 - Our interpreter language is block-structured (Assignment 2)
 - ► C, C++, Java are block-structured languages
- To implement nesting, each environment can have a pointer to its "parent" environment, i.e., the environment representing the enclosing scope

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ 三三 - のへぐ



▲□▶ ▲□▶ ▲三▶ ▲三▶ 三三 のへで



▲□▶ ▲□▶ ▲三▶ ▲三▶ 三三 のへで



d;



▲□▶ ▲□▶ ▲三▶ ▲三▶ 三三 のへで



▲□▶ ▲□▶ ▲三▶ ▲三▶ 三三 のへで





・ロト・日本・日本・日本・日本・日本





◆□ ▶ ◆□ ▶ ◆ □ ▶ ◆ □ ▶ ◆ □ ● ◆ ○ ○ ○



▲□▶▲□▶▲≡▶▲≡▶ ≡ のへ⊙





▲□▶ ▲圖▶ ▲≣▶ ▲≣▶ = 差 = 釣�?









In a language where every variable's scope is known statically, we can use *lexical addresses* to associate variable references with their definitions

Each variable has an integer position

Lexical address is pair (*depth*, *position*)

depth: 0 if referenced variable is in current environment, 1 if in parent, 2 if in grandparent, etc.

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ ▲ 三 ● ● ●



◆□▶ ◆□▶ ◆ □▶ ◆ □▶ ● □ ● ● ●

var a; var b; var c; a = 1;b = 2;c = 3;function add1(a) { var b; b = 1;c = 4;a + b;} var d; d = add1(c);println(a); println(b); println(c); d;



◆□ ▶ ◆□ ▶ ◆ □ ▶ ◆ □ ▶ ● □ ● ● ● ●







Analyze source:

- Keep track of current (static) environment, initially the global environment
- ► Enter a nested scope → enter a nested environment (with previous environment as its parent)
- \blacktriangleright Leave a nested scope \rightarrow return to parent environment
- Keep track of names defined (variables, functions)
- As long as definitions precede uses, we can associate each reference to a name with an entry in a static environment
- Static (pre-execution) environments are also called symbol tables
 - Much more about these when we move on to compilers!

Runtime data structures

・ロト・4回ト・ミト・ミト・ミージへ()

Important runtime data structures to support the execution of the program being interpreted:

▲□▶ ▲圖▶ ▲ 臣▶ ▲ 臣▶ ― 臣 … のへぐ

- Values
- Environments
- Functions

How to represent a runtime value? Assuming a dynamically-typed language, where a value's data type can't (necessarily) be predicted until the program runs, we need:

▲□▶ ▲□▶ ▲□▶ ▲□▶ ■ ●の00

- ► The value's type
- A representation of the value

- Different data types require different representations
- Some values are fixed size (e.g., fixed-precision integers, floating point values, pointer or reference to an object)
- Some values require arbitrary storage (e.g., arrays, objects, etc.)
- Typical approach: allow the representation to be either an "atomic" (fixed-size) value, or a pointer to a "dynamic" representation object

```
enum ValueKind {
  // "atomic" values
  VALUE_INT,
  VALUE DOUBLE,
  VALUE INTRINSIC FN,
  // other kinds of atomic values...
  // "dynamic" values
  VALUE_FUNCTION,
  VALUE ARRAY,
  VALUE_STRING,
  // other kinds of dynamic values
};
```

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

```
union Atomic {
    int ival;
    double dval;
    IntrinsicFn intrinsic_fn;
    // etc.
};
```

IntrinsicFn is a pointer to an "intrinsic" function, i.e., one implemented directly by the interpreter

◆□▶ ◆□▶ ◆ □▶ ◆ □▶ ● □ ● ● ●

Value

```
class Value {
private:
  ValueKind m_kind;
union {
    Atomic m_atomic;
    ValRep *m_valrep;
};
```

public:

```
// various constructors...
Value(const Value &other);
~Value();
```

Value &operator=(const Value &rhs);

▲□▶ ▲□▶ ▲ □▶ ▲ □▶ □ のへぐ

```
// member functions...
};
```

All classes implementing representations of dynamic values (functions, arrays, strings, etc.) derive from ValRep. I.e.:

```
class Function : public ValRep {
   // ...
};
class Array : public ValRep {
   // ...
};
class String : public ValRep {
   // ...
};
```

The type of representation object is indicated by the Value's m_kind value: e.g., if it's VALUE_FUNCTION then m_valrep points to a Function object

- The Value class has (and needs) value semantics (copy constructor and assignment)
- Runtime values are frequently copied from one variable to another, passed to a function, returned from a function, etc.
- How do we ensure that dynamic representation objects are deallocated when no longer needed?
 - Issue: multiple Value instances might have pointers to the same dynamic representation object

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ ▲ 三 ● ● ●

More on this in a bit

```
function add1(x) {
    x + 1;
}
function apply(f, v) {
    f(v);
}
var g;
g = add1;
apply(g, 4);
```

The above program computes the value 5. Many dynamic languages and all functional languages treat functions as values.

An environment is

- A map of names to values
- A reference (pointer) to the parent environment (representing the enclosing scope)
 - ▶ The environment representing the global scope does not have a parent

```
class Environment {
private:
   Environment *m_parent;
   std::map<std::string, Value> m_varmap;
```

◆□▶ ◆□▶ ◆三▶ ◆三▶ 三三 のへぐ

```
public:
    Environment(Environment *parent);
    ~Environment();
```

```
// member functions...
};
```

Operations an environment should support:

- Creating a new variable (setting it to some initial default value)
 - It should be an error to define a variable that already exists
- Determining whether a variable is defined locally in the environment
- Looking up the value of a locally-defined variable
- Looking up the value of a variable, including searching outer scope(s) if necessary

A function is

- ► A list of 0 or more parameters
- A pointer to the environment representing the scope enclosing the function (for top-level functions, the global scope)
- An AST representing the body of the function

Since a function is a dynamic value, its representation is derived from ValRep.

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

```
class Function : public ValRep {
private:
  std::vector<std::string> m_params;
  Environment *m_parent_env;
  Node *m body;
public:
  Function(const std::vector<std::string> &params,
           Environment *parent_env,
           Node *body);
  ~Function():
     member functions...
  11
};
```

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

As we've seen, executing a function call means:

- Creating a new environment for it (with global environment as parent)
- Evaluating argument expressions
- Binding function parameters to argument values in the new function call environment
- Evaluating the body of the function in the new function call environment
 - Because the body of a function is a block, it will have its own environment whose parent is the function call environment
- Result of evaluating body is result of function
 - Becomes value of function call expression at call site

Reference counting

◆□ > ◆□ > ◆ 三 > ◆ 三 > ● ○ ○ ○ ○

Reference counting is a simple and mostly-effective way of keeping track of uses of dynamically allocated objects, and deleting them when they are no longer needed.

The idea is to maintain an integer *reference count* within each dynamic object:

- ▶ If the reference count is > 0, it is still in use
- If the reference count is = 0, it is no longer in use, and should be deleted

▲ロ ▶ ▲周 ▶ ▲ 国 ▶ ▲ 国 ▶ ● の Q @

C++ makes it easy to track reference counts using "smart pointer" objects.

Rather than the interpreter keeping direct pointers to dynamic objects (ValRep *), it wraps them in an object with value semantics that

- ▶ increments and decrements reference counts as needed
- ▶ frees dynamic objects when the reference count reaches 0

In Assignment 2, the Value class serves as the smart pointer type for dynamic objects. (Which makes sense, because Value represents a runtime value.)

The important operations for the smart pointer type are *attach* and *detach*:

- Attach: increment the dynamic object's reference count and store a pointer to it
- Detach: decrement current dynamic object's reference count, delete it if the reference count is 0
 - For safety, it's not a bad idea to store null in the pointer so that there isn't a dangling pointer to the dynamic object

▲□▶ ▲□▶ ▲ 三▶ ▲ 三▶ ▲ 三 ● ● ●

- Constructor from pointer to dynamic object: attach to the dynamic object
- Copy constructor: attach to the other smart pointer's dynamic object (if there is one)
- Destructor: detach from current dynamic object (if there is one)
- Assignment operator: detach from current dynamic object (if there is one), attach to other smart pointer's dynamic object (if there is one)

- Reference counting has trouble reclaiming dynamic objects if there are reference cycles
 - E.g., object A has a pointer to object B, and object B has a pointer to object A
 - Both A's and B's reference counts stay at 1 even though they are not reachable

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□ ● ● ●

► Various solutions exist: for example, periodically run a garbage collector