

CS 162 Week 9

Kyle Dewey

Overview

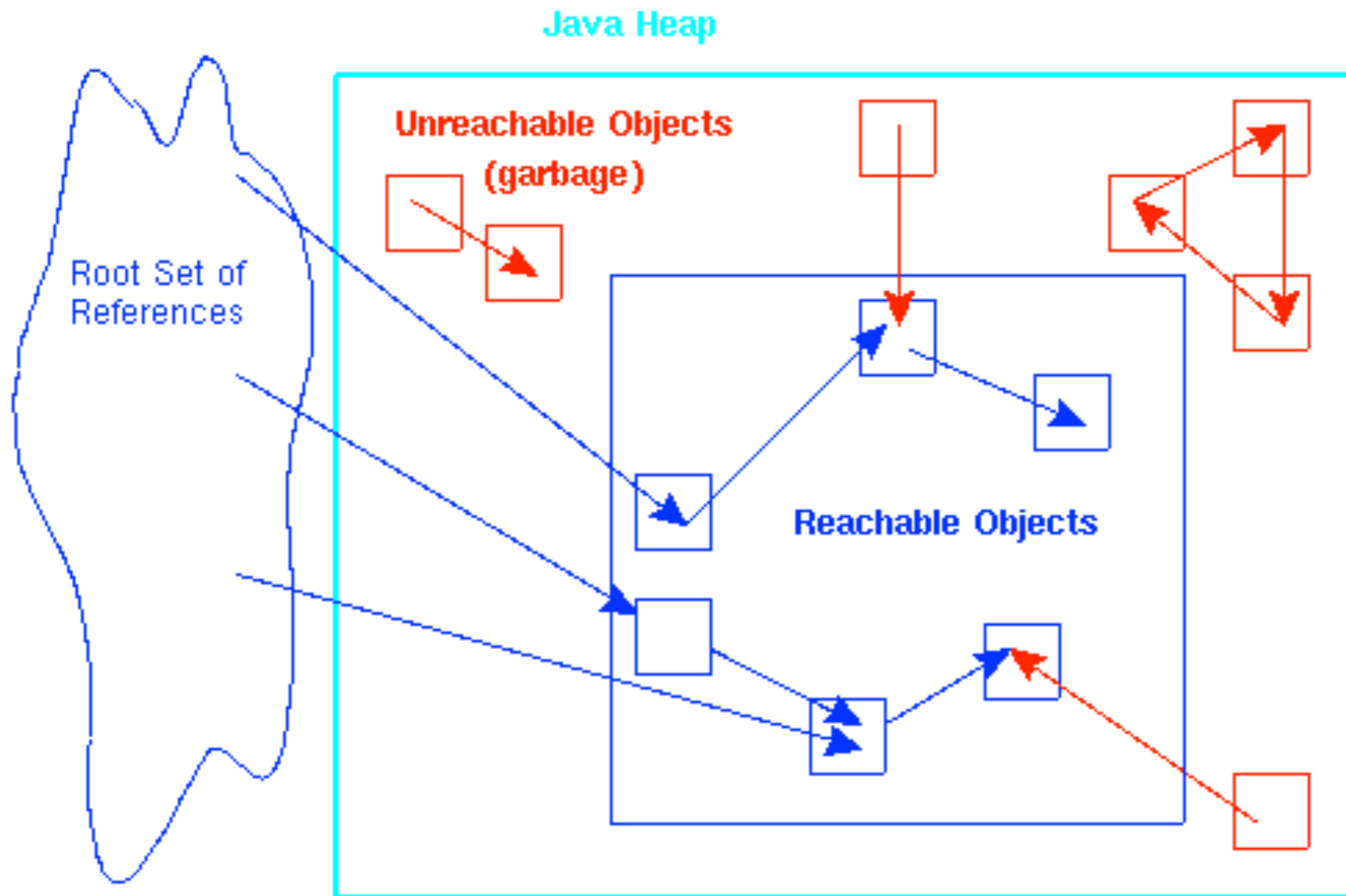
- What needs to be done
- Quirks with GC on miniJS
- Implementing GC on miniJS

The Plan

- Implement two garbage collectors:
 - Semispace
 - Mark/sweep

GC Reachability

- Reachability means we can access the object



Liveness

- If we can reach an object from the **root set**, then the object is live
- If we cannot reach it, then it is dead
- Reclaim only dead objects

Question

- The typical root set consists of the values of variables on the stack
- What is the root set for miniJS?

```
var a, obj in
  obj := {foo: "bar"};
  obj.bar := obj.foo;
  a := 12;
  obj := {b: a, o: obj}
```

Issue #1: Variables on the Stack

- miniJS does not have a usual stack
- Variable values tracked by recursive calls to eval with InScope objects

```
case Let( xs, t ) =>
{
  val bindings = xs map ( _.x → ( σ += UndefV() ) )
  inScope( ρ ++ bindings ) eval t
}
```

Solving Issue #1

- Introduce a global mutable stack that acts as the environment
- Variable to address bindings are pushed onto it in the same way as seen with a usual runtime stack

Original

```
case Let( xs, t ) =>
{
  val bindings = xs map ( _.x → ( σ += UndefV() ) )
  inScope( ρ ++ bindings ) eval t
}
```

New

```
case Let( xs, t ) =>
{
  val addrs = addBindings(xs.map(_.x → UndefV()))
  val retval = eval(t)
  ρ.popTimes(addrs.size)
  retval
}

def addBindings(bindings: Seq[(String, Storable)]): Seq[Address] =
  bindings.map(pair => {
    val ref = (σ += pair._2)
    ρ.push(pair._1 → ref)
    ref
  })
```

Issue #2: Intermediate Values

- What's problematic with this for GC?

```
case Update( e1, e2, e3 ) =>
{
  val adr = eval( e1 )
  val fld = eval( e2 )
  val rhs = eval( e3 )

  (adr, fld) match {
    case (adr:Address, fld:StrV) =>
      {
         $\sigma(\text{adr}) = \text{toObj}(\sigma(\text{adr})) \text{ :+ } (\text{fld} \rightarrow \text{rhs})$ 
        UndefV()
      }
    case _ => throw undefined
  }
}
```

Issue #2: Intermediate Values

- We can operate on store-allocated values without having a binding in the environment

```
( {foo: "bar"} ).baz := { "temp": 1 }
```

Solving Issue #2

- Put all temporary variables into the root set

```
case Update( e1, e2, e3 ) =>
{
  val adr = eval( e1 )
  RootSet.pushExtra(adr)
  val fld = eval( e2 )
  RootSet.pushExtra(fld)
  val rhs = eval( e3 )
  RootSet.pushExtra(rhs)

  (adr, fld) match {
    case (adr:Address, fld:StrV) =>
      {
         $\sigma(\text{adr}) = \text{toObj}(\sigma(\text{adr})) \text{ :+ } (\text{fld} \rightarrow \text{rhs})$ 
        RootSet.popExtraTimes(3)
        UndefV()
      }
    case _ => throw undefined
  }
}
```

Getting the Root Set

- Calling `RootSet ()` will get the global root set
- This root set is a set of `Storable`, not `Address` as in the usual definition

Representing the Heap

```
class Heap(size: Int) extends HeapInterface {  
  protected val heap = new Array[Any](size)  
}  
  
// blocksAllocated includes the metadata  
case class AllocatedMetadata(blocksAllocated: Int, typ: StorableType)  
  
// blocksAvailable includes the metadata  
case class FreeMetadata(blocksAvailable: Int, next: Int)
```

- HeapInterface provides functions for reading / writing objects
- See the StubCollector for detailed information on usage

Heap With StubCollector

- After allocating the number 1:

Allocated Metadata Size: 2 Type: NumV	NumV(1)			

Heap With StubCollector

- After allocating the number 1 and a closure:

Allocated Metadata Size: 2 Type: NumV	NumV(1)	Allocated Metadata Size: 4 Type: CloV	<<Variable Names>>	<<Closure Term>>
<<Closure Environment >>				

Heap With StubCollector

- After allocating the number 1, a closure, and the string “foo”:

Allocated Metadata Size: 2 Type: NumV	NumV(1)	Allocated Metadata Size: 4 Type: CloV	<<Variable Names>>	<<Closure Term>>
<<Closure Environment >>	Allocated Metadata Size: 2 Type: StrV	StrV(“foo”)		

Heap With StubCollector

- After allocating the number 1, a closure, the string “foo”, and the boolean true:

Allocated Metadata Size: 2 Type: NumV	NumV(1)	Allocated Metadata Size: 4 Type: CloV	<<Variable Names>>	<<Closure Term>>
<<Closure Environment >>	Allocated Metadata Size: 2 Type: StrV	StrV(“foo”)	Allocated Metadata Size: 2 Type: BoolV	BoolV(true)

The Collectors

- Two key functions: `gcAlloc` and `gcRead`
- These do exactly what their names suggest

```
def gcAlloc(s: Storable): Address  
def gcRead(a: Address): Storable
```

StubCollector

Mutation in miniJS

- Old semantics: update the `Storable` at a given address to be some new `Storable`
- What's wrong with this with respect to the new heap?

```
case Assign( Var(x), e ) =>
{
  val v = eval( e )
   $\sigma( \rho( x ) ) = v$ 
}
```

Mutation Issue

- Problem: different objects take up different lengths
- Since miniJS is dynamically typed, we could switch the kind of object stored
- If we want to store a new object that's bigger than what the old one took up, we generally won't have the space at the same address

Mutation Issue

```
var a, b in
  a := 1;
  b := 2;
  a := () => {output b}
```

a

Allocated Metadata Size: 2 Type: NumV	NumV(I)			
--	---------	--	--	--

Mutation Issue

```
var a, b in
  a := 1;
  b := 2;
  a := () => {output b}
```

a

b

Allocated Metadata Size: 2 Type: NumV	NumV(1)	Allocated Metadata Size: 2 Type: NumV	NumV(2)	
--	---------	--	---------	--

Mutation Issue

```
var a, b in
  a := 1;
  b := 2;
  a := () => {output b}
```

Closures take up 4 units, but the original address has only 2

a

b

Allocated Metadata Size: 2 Type: NumV	NumV(1)	Allocated Metadata Size: 2 Type: NumV	NumV(2)	
--	---------	--	---------	--

Handling Mutation

- Instead of trying to reuse addresses, we allocate to a new address
- We update the old address to have the same location as the new address

```
def gcModify(a: Address, v: Storable) {  
  // done via emulation - alloc again and update the address  
  RootSet.pushExtra(v)  
  val newAddr = gcAlloc(v)  
  RootSet.popExtra()  
  a.loc = newAddr.loc  
}
```

Handling Mutation

- For this update to work, your interpreter must have the following invariant: at any point in time, there is at most one `Address` object associated with each underlying position in the heap
- If addresses are only made in `gcAlloc` for freshly allocated values, this will be guaranteed
- Without this, we can still have `Addresses` that point to the old address after update

Other Assorted Notes

Tracing

- Be sure to look at `values.scala`
- Extra kinds of `Storables` have been added
 - This `miniJS` has lists
 - `ObjectVs` are implemented internally with lists that extend `Storable`
- You need to be able to trace these lists

trace

- A special `trace` function is provided, which will simply print the string specified if the `-trace` flag is set
- Very useful for debugging
- If there is inadequate tracing, you will be penalized

assert

- For any assumptions you have, you should make sure they are true with the `assert` statement
- Many bugs will not trigger a typical error until long after they occurred, and proper usage of `assert` can help shorten this gap
- I.e. `assert` can cause bugs to reveal themselves sooner than usual

freelist.scala
and gc.scala