Concurrent Objects in Orc

Arthur Peters

Motivation

Orc with concurrent objects reinterprets traditional object-oriented programming through the lens of a concurrent language. Concurrent programming is required due to both multiprocessing architectures and asynchronous interactions with the outside world. Object-oriented programming is a powerful abstraction and reuse technique. In the past, concurrency has been understood in the context of objects. The goal of Orc’s concurrent objects is to understand objects in the context of concurrency.

Concurrent objects will be useful in many applications including:
- GUIs
- Websites
- Home automation

Related Models

- Thread-safe Sequential Objects (such as monitors in Java)
- Active Objects (from Nierstrasz and others)
- Actors (as used in Erlang)

All of these models are specifically designed to prevent concurrency within a single object or actor. As such they confine concurrency control and data/computation structure.

Goals

- Pervasively concurrent by default even inside objects
  - Eager: Start computation as soon as possible
  - Lenient: Allow computation to begin before its inputs are computed
  - Programmer removes concurrency instead of adding it

Examples

class FaninChannel(val channels) extends Source {
  -- Build an internal channel
  val outchannel = Channel()
  -- Setup an ongoing computation that transfers values on
  -- input channels to the output channels
  val _ = each(channels) >>
    repeat(c.get() >> outchannel.put(x()))
  -- Allow users to get values from the output channel
  def get() = outchannel.get()
}

Fractal Explorer

{}  
-- Initialize Objects
val renderEvent = FanoutChannel()
val button = Button("Render")
val canvas = Canvas(size, size)
val frame = Frame([button, canvas])

-- When the button is pressed, send a render event
button.onAction() >> renderEvent.put(CurrentConfig()) >> stop |
-- When a render is requested render, but stop if another
-- render is started
renderEvent.listen() >> |
  RenderScene(canvas, c) | renderEvent.listen() |
| >> stop |
-- When the canvas is clicked, reconfigure the scene and render
canvas.onClicked() >> event |
  renderEvent.put(NewConfig(event)) >> stop |
-- When the frame is closed, kill this whole program
frame.onClosing() |

Computation and Interaction within Objects

Features Provided by Models

<table>
<thead>
<tr>
<th>Thread-safe Objects</th>
<th>Fields</th>
<th>Background Computation</th>
<th>Methods</th>
<th>Background Conc.</th>
<th>Decouples Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Objects</td>
<td>Sync</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Actors</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Objects</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* Sync means that the feature is supported, but synchronously.

Object Design

Concurrent objects are pervasively concurrent but still share many of the features of traditional objects found in languages such as Scala.

- Classes contain methods and fields
- Classes can inherit from and mixin superclasses
- All fields in a class are evaluated in parallel
- Access to a field blocks until the field is evaluated
- Superclass fields are still evaluated concurrently with the overriding subclass field

Objects execution can be controlled using the same tools used for any Orc expression.

Examples

class FanoutChannel {
  -- A lock to protect the listener list
  val lock = Semaphore(1)
  -- A list of output ("Listener") channels
  val listeners = Ref([])

  -- Put a value on every listener channel
  def put(v) = each(listeners?) >> c.put(v)
  -- Return a listener channel that will be active until this call is killed
  def newListener() = withLock(lock, { 
    val c = Channel()
    runOnKill({ removeListener(c) }) >> listeners :> c : listeners? >> c
  })
  -- Remove a listener channel
  def removeListener(c :: Channel) = withLock(lock, { 
    listeners := filter(listeners_, { _ /= c })
  })

  -- Publish all values put on this channel while this call is active
  def listen() = 
    val c = newListener()
    repeat(c.get())
}
