Lightweight Language Support for Type-Based, Concurrent Event Processing

Philipp Haller
EPFL
Scala Workshop 2010
Motivation

- Concurrent and distributed programming with asynchronous events is indispensable
- Many applications: event processing, web applications, algorithmic trading, ...
- In Scala: actors via embedded DSL
- **Problem:** bad performance of innocent code patterns
- **Idea:**
  - Selectively enrich run-time type information to avoid performance hazards
Actors in Scala

- Actors: processes that exchange messages
  
  ```scala
  actor ! message // message send
  
  react { // message receive
    case `msgpat`i => action1
    ...
    case `msgpat`n => actionn
  }
  
  - Send is *asynchronous*: messages are buffered in actor's *mailbox*
  
  - `react` waits for next message that matches any of the patterns `msgpat`i
Simple buffer actor:

```java
loop {
  react {
    case Put(x) =>
      react {
        case Get(from) =>
          from ! x
      }
  }
}
```

Scenario:
- Lots of Put messages in mailbox
- Get messages arrive slowly
  - Outer react *finishes quickly*
  - Inner react *searches entire mailbox* in most cases
  - **Worst case:** for every Get message: go through all Put messages in mailbox!
Optimization by Hand

```scala
val putQ = new Queue[Int]
val getQ = new Queue[Actor]
loop {
  react {
    case Get(from) =>
      if (putQ.isEmpty)
        getQ.enqueue from
      else
        from! Put(putQ.dequeue)
    
    case Put(x) =>
      if (getQ.isEmpty)
        putQ.enqueue x
      else
        getQ.dequeue! Put(x)
  }

  explicit queues replace actor's mailbox
}
```
Partial Functions in Scala

```scala
{  
case msgpat_1 => action_1  
  ...
  case msgpat_n => action_n  
}
```

is compiled to
(anonymous) class that extends

```scala
trait PartialFunction[-A, +B] extends Function1[A, B] {  
def isDefinedAt(x: A): Boolean }

trait Function1[-A, +B] {  
def apply(x: A): B }
```
Implementing `react`

```scala
... react
{
  case Get(from) => // handle msg
}

def react(handler: PartialFunction[Msg, Unit]) = {
  mailbox.extractFirst(handler.isDefinedAt) match {
    case None => waitingFor = handler; suspendActor()
    case Some(msg) => handler(msg)
  }
}
```

“Blindly” tests each message in mailbox

simplified...
Idea: Reify Matched Types

1. Add method to `PartialFunction[A, B]`:
   ```scala
def definedFor : Array[Class[_]]
```

2. Split mailbox into subqueues:
   ```scala
val mailbox : Map[Class[_], Queue[Msg]]
```

3. `extractFirst(handler)` skips uninteresting subqueues
   - Only search queue `mailbox(clazz)` if `handler.definedFor` contains `clazz`
Translucent Functions

```
trait TranslucentFunction[-A, +B]
  extends PartialFunction[A, B] {
    def definedFor: Array[Class[_]]
  }
```

- Each class in `definedFor` represents a case class that is the type of a pattern, e.g.

  ```scala
  abstract class Msg
  case class Put(x: Int) extends Msg
  case class Get(from: Actor) extends Msg

  val tf = { case Put(y) => y }
  ```

- `tf` has type `TranslucentFunction[Msg, Int]` and `tf.definedFor == Array(classOf[Put])`
Applying Translucency to Actors

• Let \( \text{fun: TranslucentFunction}[A, B] \) such that
  - \( \text{fun.definedFor} == \text{Array}(C_1, \ldots, C_n) \)
  - \( \text{typeof}(\text{msg}) == \text{Msg} \) for all \( \text{msg} \) in \( \text{subqueue} \)
  - \( \text{Msg} <: C_i \) for all \( i = 1..n \)

• We want to conclude that

\[
\text{fun.isDefinedAt msg} == \text{false} \text{ for all msg in subqueue}
\]
Getting definedFor Right: Subtyping

```scala
abstract class Msg
case class Put(x: Int) extends Msg

mailbox = Map(classOf[Msg] -> Queue(),
               classOf[Put] -> Queue(Put(42)))

react { case any: Msg => ... }
```

- Problem: will not find `Put(42)` if only `mailbox(classOf[Msg])` is searched!
- Solution: `definedFor` is empty if a pattern type is not subtype of a case class
- All subqueues searched if `definedFor` empty
Getting `definedFor` Right: Modularity

- Consider usage of translucent function
  
  ```scala
  react { case Put(x) => ... }
  ```

- Adding separately compiled
  
  ```scala
  class PutTwice(x: Int) extends Put(x)
  ```

- Problem: find `PutTwice` messages without recomiling translucent function!
  
  - Adding a separately compiled class should not affect `definedFor`

- Solution: `definedFor` contains only case classes

Assume no case class inheritance
**Definition 1** (Invariant of Translucent Functions).

If \( f : \text{TranslucentFunction}[A, B] \) and 
\( f.\text{definedFor} \neq \text{Array}() \), then 
\[
  f.\text{isDefinedAt}(o) \Rightarrow \text{typeof}(o) \leq C \text{for some}
\]
\[
  \text{case class } C \text{ such that } \text{classOf}[C] \in f.\text{definedFor}
\]
Optimizing Actors

- **Foundation:**
  
  \[
  (f.definedFor \neq Array() \land \forall \text{classOf}[C] \in \text{definedFor}. \neg \text{typeof}(o) <: C') \Rightarrow \neg f.\text{isDefinedAt}(o)
  \]

- **Split mailbox:** using case classes \( C_1, \ldots, C_n \)
  
  - if \( \text{typeof}(msg) <: C_1 \) then \( \text{typeof}(msg) \not<: C_i \) (\( i \neq 1 \))

- Insert msg such that \( \text{typeof}(msg) <: C_i \) into subqueue for \( C_i \); into global queue otherwise

- search only subqueues for classes in \( \text{definedFor} \)
Optimizing Join Patterns

```
join {
  case Exclusive(from) => join {
    case Sharing(0) => from ! OK
  }
  case ReleaseExclusive(from: Reactor) => self ! Sharing(0); from ! OK
  case Shared(from: Reactor) => join {
    case Sharing(n) =>
      self ! Sharing(n+1); from ! OK
    }
    case ReleaseShared(from) => join {
      case Sharing(n) if n > 0 =>
        self ! Sharing(n-1); from ! OK
    }
  }
}
```

- Extending partial match requires matching messages received so far against next pattern
- **Skip subqueues** with messages that cannot match
Implementation

- Small extension to Scala 2.8 compiler
- Partial function literals are translucent:

  ```scala
def react(fun: TranslucentFunction[Msg, Unit]) = ... react { case Put(x) => ... }
```

- Add `def definedFor: Array[Class[_]]`, populated with `className[C]` for each case class C such that the type of a pattern is a subtype of C
- `definedFor` is empty iff one of the pattern types is not subtype of a case class
Implementation (2)

● Drop-in replacement for actors message queue

● Incremental queue splitting
  – Unknown class in `definedFor`? Create new subqueue and populate with conforming messages from global queue (operate on `queue nodes`)

● Queue nodes contain `time stamps to maintain ordering` across subqueues

● Cache mappings between concrete message classes and target subqueues (for superclass)
Experiments

• Worst-case overhead in chameneos-redux
  – No nested/sequenced receives: *translucent functions only incur overhead!*

<table>
<thead>
<tr>
<th>Nested receives</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala 2.8.0</td>
<td>Yes</td>
</tr>
<tr>
<td>ActorFoundry</td>
<td>Yes</td>
</tr>
<tr>
<td>Akka 0.6</td>
<td>No</td>
</tr>
<tr>
<td>translucent</td>
<td>Yes</td>
</tr>
</tbody>
</table>

• Baseline 18% slower than ActorFoundry
• 23% overhead compared to baseline
Experiments (2)

- Producer/consumer scenario

<table>
<thead>
<tr>
<th>Impl./time[ms]</th>
<th>20000</th>
<th>200000</th>
<th>2000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala 2.8.0, default</td>
<td>3102</td>
<td>387669</td>
<td></td>
</tr>
<tr>
<td>Scala 2.8.0, explicit</td>
<td>166</td>
<td>1693</td>
<td>16894</td>
</tr>
<tr>
<td>translucent</td>
<td>262</td>
<td>1931</td>
<td>16461</td>
</tr>
<tr>
<td>translucent-explicit</td>
<td>305</td>
<td>1745</td>
<td>18241</td>
</tr>
</tbody>
</table>

- Default does not scale (quadratic factor!)
  - Affects all other actor implementations!

- Overhead compared to manual optimization shrinks from 58% (20,000) to 14% (200,000)
Code Size

• No execution overhead when used in place of partial functions; class files get bigger, though

  → Can we make all partial functions translucent?

• Only very small increase in code size
  – Generated class files for compiler and standard library: increase by 0.26% or 140 KB

• Actor-based benchmark code:
  – chameneos-redux: 3.7% increase
  – producer-consumer: 8.9% increase
Conclusion

- Minimal compiler extension (not syntax)
- Refinement of Scala's partial functions
- Potential for significant performance improvements of concurrency abstractions
- Future work: apply lessons learned in compiler plug-in for optimizing join patterns
- Questions?