Lightweight Language Support for Type-Based, Concurrent Event Processing

Philipp Haller
EPFL

Scala Workshop 2010

Motivation

- Concurrent and distributed programming with asynchronous events is indispensible
- 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

```
actor ! message                 // message send

react {                         // message receive
  case $msgpat_1$ => $action_1$
  ...
  case $msgpat_n$ => $action_n$
}
```

- Send is *asynchronous:* messages are buffered in actor's *mailbox*
- `react` waits for next message that matches any of the patterns $msgpat_i$

Example

Simple buffer actor:

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

ent

Loo

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

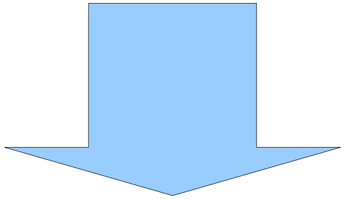
```
val putQ = new Queue[Int]
val getQ = new Queue[Actor]
loop {
  react {
    case Get(from) =>
      if (putQ.isEmpty)
        getQ enqueue from
      else
        from ! Put(putQ.dequeue)

    case Put(x) =>
      if (getQ.isEmpty)
        putQ enqueue x
      else
        getQ.dequeue ! Put(x)
  }
}
```

Explicit queues **replace** actor's mailbox

Partial Functions in Scala

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



is compiled to (anonymous) class that extends

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

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

Implementing react

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

“Blindly” tests each message in mailbox

simplified...

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

Idea: Reify Matched Types

1. Add method to `PartialFunction[A, B]`:

```
def definedFor : Array[Class[_]]
```

2. Split mailbox into subqueues:

```
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.

```
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 `fun: TranslucentFunction[A, B]` such that
  - `fun.definedFor == Array(`C_1`, ..., `C_n`)`
  - `typeof(msg) == Msg` for all `msg` in `subqueue`
  - `Msg` ~~`<:`~~ C_i for all i = 1..n
- We want to conclude that

`fun isDefinedAt msg == false` for all `msg` in `subqueue`

Getting `definedFor` Right: Subtyping

```
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

```
react { case Put(x) => ... }
```

- Adding separately compiled

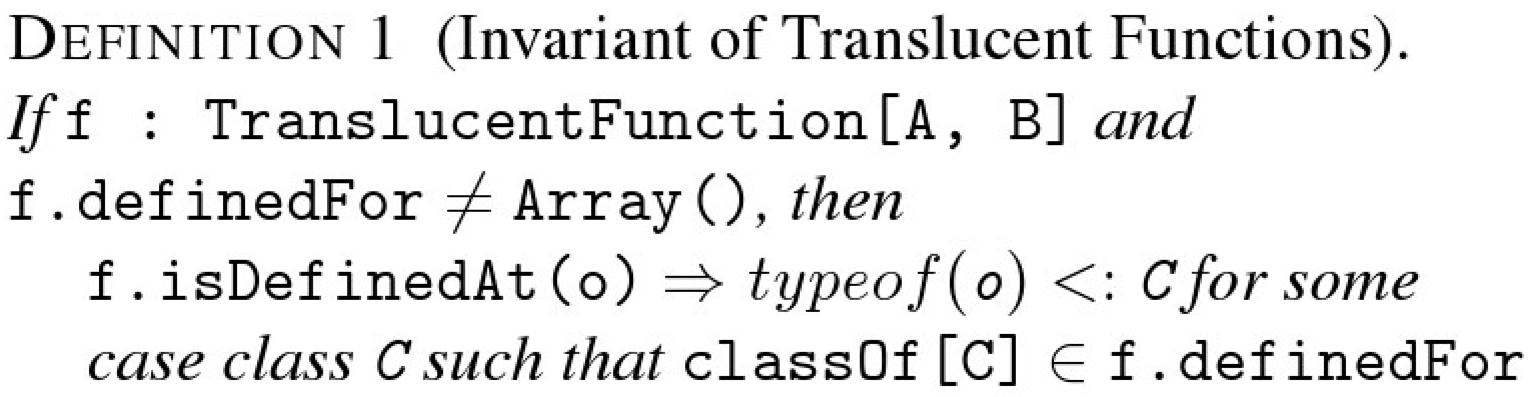
```
class PutTwice(x: Int) extends Put(x)
```

- Problem: find `PutTwice` messages w recompiling translucent function!
  - Adding a separately compiled class should not affect `definedFor`

Assume no case class inheritance

- Solution: `definedFor` *contains only case classes*

Translucent Functions, Precisely



DEFINITION 1 (Invariant of Translucent Functions).
If `f : TranslucentFunction[A, B]` *and*
`f.definedFor` $\neq$ `Array()`, *then*
  `f.isDefinedAt(o)` $\Rightarrow$ $typeof(o) <: C$ *for some*
  case class C *such that* `classOf[C]` $\in$ `f.definedFor`

Optimizing Actors

- Foundation:

$$(\texttt{f.definedFor} \neq \texttt{Array()} \wedge$$
$$(\forall\, \texttt{classOf[C]} \in \texttt{definedFor}\,.\, \neg \mathit{typeof}(\texttt{o}) <: C)) \Rightarrow$$
$$\neg\, \texttt{f.isDefinedAt(o)}$$

- Split mailbox: using case classes C_1, ..., C_n
  - if typeof(msg) <: C_1 then typeof(msg) $\not<:$ C_i $(i \neq 1)$
- Insert msg such that typeof(msg) <: C_i into subqueue for C_i; into global queue otherwise
- search only subqueues for classes in `definedFor`

Optimizing Join Patterns

```
join {
  case Exclusive(from) => join {
    case Sharing(0) => from ! OK
  }
  case Re
    self
  case Sh
    case
      sel
  }
  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:

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

- Add `def definedFor: Array[Class[_]]`, populated with `classOf[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!*

	Nested receives	Time [ms]
Scala 2.8.0	Yes	11151
ActorFoundry	Yes	9435
Akka 0.6	No	8065
translucent	Yes	13731

- Baseline 18% slower than ActorFoundry
- 23% overhead compared to baseline

Experiments (2)

- Producer/consumer scenario

Impl./time[ms]	20000	200000	2000000
Scala 2.8.0, default	3102	387669	
Scala 2.8.0, explicit	166	1693	16894
translucent	262	1931	16461
translucent-explicit	305	1745	18241

- 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?