Lightweight Language Support for Type-Based, Concurrent Event Processing

Philipp Haller
EPFL, Switzerland
firstname.lastname@epfl.ch

Abstract
Many concurrent applications are structured around type-based event handling. Scala provides a library for event-based actors, which allows common idioms to be expressed in a concise and intuitive way. However, innocent-looking programs can exhibit catastrophic performance under certain conditions. In this paper, we introduce translucent functions, a type-based refinement of Scala’s pattern-matching functions. Translucent functions additionally provide the runtime types of classes that identify disjoint cases in a pattern. We show how this additional type information can be used to optimize actors as well as a form of join-style synchronization.

1. Introduction
Concurrent applications are often structured as concurrent components processing synchronous or asynchronous messages or events. These events are processed depending on their value, their type [7], or both.

Scala contains an actor library [12] for event-based programming. Actors [2, 14] are concurrent processes exchanging asynchronous messages. Every actor has a mailbox, which buffers incoming messages that have not been processed, yet. Event-based handling of messages is done using the react primitive [13], which tries to remove the earliest message from the current actor’s mailbox that matches one of a provided set of patterns. It has the following form:

```scala
react {
  case msgpat1 => action1
  ...
  case msgpatn => actionn
}
```

The first message that matches any of the patterns `msgpat_i` is removed from the mailbox, and the corresponding `action_i` is executed. If no pattern matches, the current actor suspends.

`react` expressions can be nested to express protocols that depend on the state of the receiver. Figure 1 shows the behavior of an actor that loops trying to receive a Put message; after that the actor receives a Get message, which carries a reference to the sender (`from`). A Get message is handled by sending the argument of the previous Put message back to the `from` actor (`from ! x`).

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

Figure 1. The logic of a simple buffer using nested reacts

Nested reacts can express a number of protocols in an intuitive way [4]. However, they may incur a very high overhead, which is not directly visible from the source code. The example in Figure 1 may perform very badly if the producer of Put messages is much faster than the actor that sends Get requests. Assume there are lots of Put messages in the buffer’s mailbox, but no Get message; this means that when the buffer waits for a Get message it has to go through the entire mailbox searching in vain for a Get message. When the next Get messages has been served, the following react finishes quickly, since a Put message can be found immediately. However, since the consumer is slow it is likely that no Get message has arrived, yet. Again, this means that the entire mailbox is searched in vain. Even when the producer stops sending Put messages, receiving a Get remains slow until “enough” Put messages have been removed from the mailbox.
Translucent Functions

In this paper we propose a lightweight, type-based refinement of pattern-matching functions, called translucent functions. A translucent function provides additional information about the types used in pattern matching clauses. The basic idea is that the compiler computes a conservative approximation of the (disjoint) types of possibly matching objects and makes this information available at run-time. An important application is type-based event handling. Translucent functions enable clients, like the actors library, to implement more efficient event matching logic by discriminating events according to the type partitions of those translucent functions that are used as filters. Using an implementation in Scala we show how examples like the above can be executed almost as efficiently as more complicated, hand-optimized variations without any changes to their code.

Related Work. Our approach is related to systems that allow pattern matching on run-time type information in functional languages [1, 17]. However, our goal is not to provide a run-time type representation of function values, but rather a type-based summary of the patterns used to define a partial function. This means that the run-time type information provided for two translucent function values with the same domain type can be different. The type information made available through translucent functions corresponds in some ways to a disjoint sum type [18]. However, our approach integrates with open type hierarchies in a modular way, where not all subtyping relationships are known at compile time. Translucent functions leverage the concept of case classes [6] in a novel way to ensure type disjointness in the presence of subtyping. The idea to split message queues according to the type model is shared with extensible pattern matching [11].

The rest of this paper is organized as follows. The following section reviews partial functions in Scala and their relationship to pattern matching. In Section 3 we introduce translucent functions as a refinement of partial functions. Section 4 explains how translucent functions can be used to optimize event handling in the context of Scala’s actors. We discuss join-style synchronization as another application in Section 4.1. We outline our implementation in Section 5, and discuss preliminary experimental results in Section 6. Section 7 concludes.

2. Partial Functions

The react primitive we have seen above is not built into the language; instead, it is a method defined in the actors library.

The pattern matching expression inside braces is treated as a first-class value that is passed as an argument to the react method. The argument’s type is an instance of PartialFunction, which is a subtrait of Function1, the trait of unary functions. The two traits are defined as follows.\footnote{For simplicity we ignore annotations for co- and contravariance.}

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

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

Functions are objects which have an apply method. Partial functions are objects which additionally have a method isDefinedAt, which tests whether the function is defined for a given argument. Both traits are parametrized; the first type parameter A indicates the function’s argument type and the second type parameter B indicates its result type.

In Scala, each pattern matching expression

```
{ case p1 => e1; ...; case pn => en }
```

is compiled into a partial function whose methods are defined as follows.

- The isDefinedAt method returns true if one of the patterns pi matches the argument, false otherwise.
- The apply method returns the value ei for the first pattern pi that matches its argument. If none of the patterns match, a MatchError exception is thrown.

3. Translucent Functions

We propose a refinement of partial functions that we call translucent functions. Translucent functions are first-class objects with the following type:

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

The idea is that a TranslucentFunction provides more information about the types of the patterns than just some upper bound A like a PartialFunction[A, B]. The definedFor method returns an array of Class instances that more precisely describe the patterns occurring in the list of cases.\footnote{The type Class[_] is a shorthand for the type Class[T] forSome { type T }, which expresses the fact that T is existentially quantified.}

Each Class instance is the run-time type
The above definition says that if \( \text{isDefinedAt} \) is defined for \( o \), then all objects for which the translucent function \( f \) is defined must have a subtype of a case class; the run-time representation of this (unique) case class is contained in \( f.\text{definedFor} \). We assume that case classes do not inherit from each other. (Currently, this is possible but deprecated in Scala 2.8; it should be easy to adapt our approach to using sealed classes instead.) This means that the types in \( \text{definedFor} \) do not overlap. The types are precise, since it is impossible to return arbitrary supertypes. For example, given the above class definitions, \( \text{definedFor} \) of the function \( \{ \text{case } B(x) => \ldots \} \) must return \( \text{Array(classOf[B])} \) or \( \text{Array()} \). The scheme supports separate compilation, since adding classes to a system never requires extending the set of types in \( \text{definedFor} \).

The above definition enables to determine whether an object is guaranteed not to be matched by a translucent function:
pattern {
  case Excl(from) => join {
    case Shar(0) => action { from ! OK() }
  }
  case RelExcl(from: Reactor) =>
    action { self ! Shar(0); from ! OK() }
  case Shared(from: Reactor) => join {
    case Shar(n) =>
      action { self ! Shar(n+1); from ! OK() }
  }
  case RelShared(from) => join {
    case Shar(n) if n > 0 =>
      action { self ! Shar(n-1); from ! OK() }
  }
}

Figure 2. A concurrent reader-writer lock using Polyphonic Scala Actors

of some pattern is not a subtype of a case class. However, it could be a case clause with a supertype S of a case class, matching arbitrary instances of subtypes of S. Therefore, we have to be conservative in this case and search all subqueues, as well as the shared queue.

Inserting a message into the mailbox proceeds as follows:

1. We acquire a time stamp for the new message.
2. We look up the message’s class to find a subqueue corresponding to a case superclass.
3. If no subqueue could be found in the previous step, we append the message to the shared queue.

To speed up both the filtering and insertion algorithms, two kinds of mappings are cached. First, when creating a new subqueue, we cache the mapping between the class and the subqueue. Second, when inserting a message into a subqueue, we cache the mapping between the message’s class and the target subqueue.

4.1 Optimizing Join-Style Actors

We also integrated translucent functions into a library that provides join calculus-style synchronization for actors. A detailed description of join patterns is beyond the scope of this paper. In the following we restrict ourselves to giving a short overview of the join library; after that we explain how its implementation can be optimized using translucent functions.

The join library we use, originally developed by Arnold deVos, is freely available on the web [19]. Figure 2 shows a concurrent reader-writer lock implemented using the library’s join combinators. The pattern combinator introduces a new set of join patterns. The patterns start just like in a react as explained above. However, join patterns can be extended to multiple messages using the join combinator in the body of the case clause. The operational effect of a join pattern is as follows. Assume the current actor is sent an Excl message (requesting to acquire the lock in exclusive write mode). In contrast to a normal react, this message will not match the first pattern, since it is joined with a Shar pattern. This means the first join pattern only matches if both Excl and Shar are present in the actor’s mailbox. Importantly, if the actor has already received Excl, it remains reactive to other join patterns as long as there is no complete, matching join pattern. Join patterns are terminated using the action combinator, defining the action to be executed once the complete pattern matches.

The basic idea of the library’s implementation is to use the react method unchanged, using nesting to receive multiple messages in sequence. As mentioned above, partially matching join patterns do not prevent other join patterns from matching. For this, the implementation keeps track of partial matches, which contain the (indices of) messages matched so far, as well as the partial function that follows in the join pattern. For instance, the partial match for the first join pattern would contain the index of a matching Excl message as well as the following partial function:

```scala
{ case Shar(0) => action { from ! OK() } }
```

When waiting for the next message, the partial functions of all partial matches are combined into a single partial function that is passed to react.

The matching logic is simple, but brute-force. Whenever a partial match could be extended by a newly received message, all messages received so far are tested against the next pattern (i.e., partial function).

Using translucent functions instead of partial functions in the join patterns, we can optimize the matching logic. First, we split the mailbox into subqueues according to the types used in pattern matches. We can do the splitting incrementally, as explained above for the case of simple actors. Second, when computing the new set of partial matches we skip messages in subqueues corresponding to types that are guaranteed to be disjoint from the types matched by the next translucent function. Our implementation of translucent functions (see Section 5) contains a version of the join library that implements this optimization. Initial experiments are encouraging. A join-based implementation of the unbounded buffer in Figure 1 shows a performance improvement of over 24x for an input of 1000 insertions/removals.

5. Implementation

We implemented translucent functions in Scala based on a recent candidate for version 2.8-Beta1. Our implementation including all benchmarks used in the following section is available in the Scala SVN repository in the translucent branch.

The implementation consists of two parts: first, we refined the transformation of function literals in positions where a type of the form TranslucentFunction[A, B] is
expected. Based on the existing logic for PartialFunctions, the type checker ensures that only function literals with case definitions conform to TranslucentFunctions. The generation of a concrete TranslucentFunction instance proceeds like in the case of a PartialFunction, which results in an anonymous class definition that includes the isDefinedAt and apply methods as explained above. For translucent functions we additionally create a private field that holds an Array[Class[[_]]] together with a public accessor method definedFor, which returns the value of this field. The array is populated using invocations of classOf[C] for each case class C such that the type of a pattern is a subtype of C.

The second part of our implementation consists of a drop-in replacement for the message queue class used in actors. Our implementation follows closely the algorithms described in Section 4. To reduce overheads, it contains specialized methods for moving messages from shared queues to subqueues, which manipulate queue nodes directly instead of the contained message objects; thereby, we avoid superfluous time stamps and extra object creations.

<table>
<thead>
<tr>
<th></th>
<th>Nested receives</th>
<th>Time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala 2.8</td>
<td>Yes</td>
<td>11151</td>
</tr>
<tr>
<td>ActorFoundry</td>
<td>Yes</td>
<td>9435</td>
</tr>
<tr>
<td>Akka</td>
<td>No</td>
<td>8065</td>
</tr>
<tr>
<td>translucent</td>
<td>Yes</td>
<td>13731</td>
</tr>
</tbody>
</table>

Table 1. Worst-case overhead of translucent functions in the chameneos-redux benchmark [21]

6. Experimental Results

In this Section we present preliminary results based on our implementation described in Section 5. We evaluate the proposed optimizations in the context of the Scala actors library [12]. All experiments were run on a dual-core Intel(R) Core 2 at 2.4GHz using Sun’s Java HotSpot Server VM 1.6.0_14 under Ubuntu 8.10 (Linux kernel 2.6.27).

As our first benchmark we ran the chameneos-redux program from the popular Computer Language Benchmarks Game [21]. We use this benchmark for two reasons: first, it allows us to evaluate our baseline performance; second, it allows us to quantify the overhead of translucent functions and mailbox splitting in a worst-case scenario. Importantly, the chameneos-redux benchmark cannot benefit from the optimizations based on translucent functions proposed in Section 4, since it never uses receive in sequence or nested. We ran the benchmark on an input of 2,000,000 measuring wall-clock execution time; in each case we took the median of 5 runs.

Table 1 compares the baseline performance of Scala actors with ActorFoundry 1.0 [15], Akka 0.6 [3] by Jonas Boner et al., and Scala actors enhanced with translucent functions and split mailboxes. ActorFoundry is a new Java-based actor implementation that compares favorably to Erlang in the chameneos-redux benchmark [15]. It uses the byte-code weaver of Kilim [20] to support very lightweight coroutine-style processes. Akka is a new Scala-based implementation of actors, which provides a different programming model, since it does not provide nested receives; this allows for a simpler and significantly cheaper execution model. The second column indicates whether the respective implementation supports nested receive expressions like Erlang-style actors [4]. This is important for two reasons: first, actors without nested receives can be implemented using a significantly cheaper model. Second, the optimizations described in Section 4 are based on nested receives.3 The third column contains the benchmark execution time in milliseconds.

<table>
<thead>
<tr>
<th></th>
<th>20,000</th>
<th>200,000</th>
<th>2,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scala 2.8, default</td>
<td>3102</td>
<td>387669</td>
<td>-</td>
</tr>
<tr>
<td>Scala 2.8, 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>

Table 2. Performance on producer-consumer benchmark

3 It is, however, possible to optimize non-nested receives by explicitly multiplexing among different translucent function values.
4 Our measurements are based on Scala 2.8-Beta1-RC8 using Reactors configured to use Doug Lea’s fork/join pool [16] implementation for JDK7.
form similarly badly. In the “explicit” configuration we implement the buffer using two explicitly managed queues for Put and Get messages, respectively, running on our baseline. This makes the code significantly more complicated while undermining the mailbox, but, as expected, this results in dramatic performance improvements. The “translucent” configuration uses again the intuitive code of “default” verbatim; the overhead over explicitly managed queues shrinks from about 58% (20,000 messages) to about 14% (200,000 messages); at 2,000,000 messages the overhead has vanished completely. The “translucent-explicit” configuration runs the code of “explicit” on our “translucent” implementation that always creates subqueues. In this case, translucent functions and mailbox splitting only incur overhead; the overheads compared to “explicit” range from about 3% (200,000 messages) to about 8% (2,000,000).

**Code size.** Translucent function objects do not add any overhead when used in place of partial functions. Therefore, it is interesting to consider extending all partial functions to include the extra information computed for translucent functions. However, doing so increases the size of the generated JVM class files, which negatively impacts class loading time (among others). However, the change in code size is very small for large, sequential Scala programs: the size of the generated class files for the Scala compiler and standard library increases by only 0.26% or 140 KB (from 54’796 KB to 54’936 KB). The impact on code size is significantly higher for our actor-based benchmark programs, since every receive operation with in-line pattern matching requires an expanded partial function class. For the chameneos-redux benchmark we measured an increase in code size of 3.7%. For the producer-consumer benchmark the code size increases by 8.9%. This is as expected, since almost all of the interesting logic is expressed in terms of partial functions in these programs.

### 7. Conclusion

In this paper we have proposed a minimal language extension of Scala called **translucent functions**. Translucent functions refine Scala’s partial functions by providing the results of a well-defined static approximation of the types of matching objects at run-time. This type information can be used to dramatically improve the performance of abstractions for type-based event handling. We expect that translucent functions also enable novel ways to efficiently implement high-level abstractions for join-style synchronization.

**Acknowledgments**

Thanks to Martin Odersky for discussions that initiated this project, and for insisting on a type-based definition of translucent functions.

**References**


