Typecasting Actors: from Akka to TAkka

Jiansen HE
University of Edinburgh
jiansen.he@ed.ac.uk

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

Philip Trinder
University of Glasgow
Phil.Trinder@glasgow.ac.uk

Abstract

Scala supports actors and message passing with the Akka library. Though Scala is statically typed, messages in Akka are dynamically typed (that is, of type `Any`). The Akka designers argue that using static types is "impossible" because "actor behaviour is dynamic", and, indeed, it is not clear that important actor support, such as supervision or name servers, can be implemented if messages are statically typed. Here we present TAkka, a variant of Akka where messages are statically typed, and show that it is possible to implement supervisors and name servers in such a framework. We show it is possible to upgrade from Akka to TAkka gradually, porting one module at a time; and we show that TAkka can support behavioural upgrade where the new message type of an actor is a supertype of the old type. We demonstrate the expressiveness of TAkka by rewriting a score of Akka applications; the percentage of lines that need to be changed varies from 44% (in a 25-line program) to 0.05% (in a 27,000-line program), with a geometric mean of 8.5%. We measure the execution speed, scalability, and throughput of TAkka programs and show that they are comparable to Akka.

1. Introduction

Can the benefits of static typing extend to message passing?

A distinguishing feature of Scala is its sophisticated static type system. A distinguishing feature of modern computing is its reliance on communication. Scala supports communication with the Akka library (Typesafe Inc. (a) 2012), partly inspired by Erlang (Armstrong 2007), which in turn is inspired by actors and message passing (Hewitt et al. 1973). However, these two features do not play well together—messages in Akka are dynamically typed (that is, of type `Any`).

There are sound reasons for this design. The Akka designers argue that using static types is "impossible" because "actor behaviour is dynamic" (Kuhn and Vlugter 2011; Kuhn et al. 2012). The design of Akka is inspired by the Erlang OTP Design Principles, including supervision trees to ensure reliability, and Erlang is dynamically typed. Since supervision is generic—a supervision framework is instantiated to processes transmitting many types of messages—it is not immediately clear whether static typing is sensible. All supervised actors need to accept a fixed set of system messages, such as Poison Pill (to kill a superised actor), and it is not clear how to incorporate these into statically typed messages. An important part of distributed infrastructure is a name server, which maps names to actors, and while it is easy to build a map from names to entities of dynamic type, it is again not clear how to build a map to entities with different static types.

This paper introduces TAkka, a variant of Akka with statically typed messages. It turns out that each of the issues above can be resolved straightforwardly. Each actor is parameterised on the type of messages it handles. In addition, each actor can also receive system messages, such as Poison Pill, which are handled by the supervision framework. The name server uses Scala "manifest" types to support a name server that maps names to actors with different static types.

It is essential that a large program can be upgraded incrementally, one module at a time, rather than requiring a monolithic change to all modules simultaneously—a principle we summarise by the motto "Evolution, not Revolution". TAkka is designed to support incremental change, and, in particular, a service written in Akka can interact with a client written in TAkka, and vice versa. Backward compatibility is supported, because TAkka actors inherit from Akka actors.

In a system with multiple components, different components may require different interfaces; since all messages are received in the same mailbox, a naive approach would be to set the type to the union of all the interfaces, causing each component to see a type containing messages not intended for it—an issue we dub the Type Pollution Problem. Fortunately, the problem may be avoided by exploiting subtyping to publish a different interface to each layer.

To evaluate the expressiveness of our system, we rewrite a score of Akka applications in TAkka. The percentage of lines that need to be changed varies from 44% (in a 25-line program) to 0.05% (in a 27,000-line program), with a geometric mean of 8.5%.

We confirm that Akka and TAkka have comparable performance in terms of throughput, runtime, and scalability. We compare the throughput of Play and Socko implementations written in both systems, and show the throughput of Akka and TAkka is on average within 8.75% of each other. We compare the runtime and scalability of six BenchErl benchmarks, and show that the runtimes of Akka and TAkka are on average within 8.89% of each other, and that they have similar scalability profiles.

The paper makes the following contributions.

- We contrast Akka and TAkka, illustrating the differences with a simple application, a supervised calculator (Section 2).
- We review the Akka API, and present the corresponding TAkka API (Section 3).

- We describe the construction of a name server that maps names to actors of different static types (Section 4).
- We demonstrate that TAkka supports the principle of "Evolution, not Revolution", by showing how Akka and TAkka code for the supervised calculator can interact (Section 5).
- We illustrate the Type Pollution Problem and its solution on an instance of the Model-View-Controller pattern (Section 6).
- We rewrite a score of Akka programs in TAkka, and measure the differences in line count (Section 7).
- We compare the throughput of Play and Socko implementations written in both Akka and TAkka, and we compare the runtime and performance of a half-dozen BenchErl benchmarks written in both Akka and TAkka (Section 8).

Section 9 concludes.

2. Actors and Their Supervision

The Akka library (Typesafe Inc. (a) 2012) implements actor programming and makes supervision obligatory. As an introduction to Akka, Figure 1 defines a calculator. Each `Actor` defines a `receive` method that reacts to incoming messages. Each `ActorRef` references an actor, and may be sent messages with the `!` operator. Our example program defines `Multiplication` and `Division` messages (lines 5–6), a `Calculator` actor which receives them (lines 8–15), and instantiates a `SafeCalculator` actor reference which is sent such messages (lines 29–35). (The relation between `Calculator` and `SafeCalculator` is explained below.)

Akka makes supervision obligatory by restricting the manner of actor creation. Calling the `actorOf` method of an actor context creates a child actor supervised by that actor, forming a tree structure (lines 23–24). There is a system-provided guardian actor which serves as the root of the supervision tree (lines 29–31). (Strictly speaking, `actorOf` is invoked on an `ActorContext` or `ActorSystem`, and it is supplied with an instance of the `Props` class to specify properties of the actor to be created; details are given in Section 3.3.) Obligatory supervision unifies the structure of actor deployment and simplifies the work of system maintenance.

The simple calculator does not consider the problematic case of dividing by 0, where an `ArithmeticException` will be raised. We define a `SafeCalculator` as the supervisor of `Calculator`. The `receive` method of the safe calculator delegates any messages received to the calculator (line 25), and defines a supervisor strategy that restarts the calculator when an `ArithmeticException` is raised (lines 17–22).

In Akka 2.0 and later versions, an undefined message (line 40) is discarded and an `UnhandledMessage` event is pushed to the event stream of the actor system. The event stream may be subscribed to by other actors who are interested in particular event messages. Our example program defines a handler (lines 44—49) and subscribes to the `UnhandledMessage` events (lines 33–35).

Figure 2 gives an equivalent TAkka implementation. Code that appears in the Akka version but not the TAkka version, or vice versa, is coloured blue. The TAkka `Actor` class takes a type parameter, `M`, which indicates the type of expected messages. Correspondingly, its message handler, `typedReceive`, has type `M => Unit`. Similarly, a type parameter is added to the `Props` class and the `ActorRef` class. An actor created from an instance of `Props[M]` has `Actor[M]`, whose corresponding actor reference has type `ActorRef[M]`. Messages sent to an actor reference of type `ActorRef[M]` must have type `M`.

Hence, developers only need to consider messages of the expected type. An attempt to send a message of a type not expected by the receiver is rejected at compile-time (line 36). TAkka uses a typed name server, as explained in Section 4. When looking up an actor by name, the expected type is provided, and if it does not match the type of the stored actor reference then an error is raised at run-time (lines 46—47). In the TAkka version, there is no need to define a handler for unexpected messages.

3. Library Design

This section elaborates the design of the TAkka library, which provides type parameters for the Actor related classes found in Akka. Figure 3 gives the Akka API, and Figure 4 gives it TAkka counterpart.

3.1 Actor

A TAkka actor has type `Actor[M]`. Unlike other actor libraries, every TAkka actor class takes a type parameter `M` which specifies the type of messages it expects to receive. The same type parameter is used as the input type of the receive function, the type parameter of the actor context and the type parameter of the actor reference pointing to itself. TAkka uses Scala `Manifest` to record type information required at runtime.

The TAkka `Actor` class inherits the Akka `Actor` trait to minimize implementation effort. Users of the TAkka library, however, do not need to use any Akka Actor API. Instead, we encourage programmers to use the typed interface given in Figure 4. The limitation of using inheritance to implement TAkka actors is that Akka features are still available to library users. Unfortunately, this limitation cannot be overcome by using delegation because, as we have seen in the `SupervisedCalculator` example, a child actor is created by calling the `actorOf` method from its supervisor's actor context, which is a private field of the supervisor. `Actor` is the only TAkka class that is implemented using inheritance. Other TAkka classes are either implemented by delegating tasks to Akka counterparts or rewritten in TAkka. Re-implementing the TAkka Actor library requires a similar amount of work as implementing the Akka Actor library.

3.2 Actor Reference

A reference to an actor of type `Actor[M]` has type `ActorRef[M]`. An actor reference provides a `!` method, through which users can send a message to the referenced actor. Sending an actor a message of unexpected type will raise an error at compile time. By using type-parameterized actor references, the receiver does not need to worry about unexpected messages, while senders can be sure that messages will be understood and processed, as long as the message is delivered.

An actor typcally responds to a finite set of different messages whereas our notion of actor reference only takes one type parameter. In a type system that supports untagged union types, no special extension is required. In a type system which supports subtyping, `ActorRef` should be contravariant on its type argument `M`, denoted as `ActorRef[-M]` in Scala. Consider the simple calculator defined in Figure 2, it is clear that `ActorRef` is contravariant because `ActorRef[Operation]` is a subtype of `ActorRef[Division]` though `Division` is a subtype of `Operation`. Contravariance is crucial to avoid the type pollution problem described in Section 6.

```scala
package sample.akka.SafeCalculator
import akka.actor.{ActorRef, ActorSystem, Props, Actor}
import akka.actor.UnhandledMessage

case class Multiplication(m:Int, n:Int)
case class Division(m:Int, n:Int)

class Calculator extends Actor {
  def receive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
  }
}
class SafeCalculator extends Actor {
  override val supervisorStrategy =
    OneForOneStrategy() {
      case _: ArithmeticException =>
        println("ArithmeticException Raised to: "+self)
        Restart
    }
  val child:ActorRef = context.actorOf(
         Props[Calculator], "child")
  def receive = {   case m => child ! m }
}
object SafeCalculatorTest extends App {
  val system = ActorSystem("MySystem")
  val calculator:ActorRef =
     system.actorOf(Props[SafeCalculator],
                    "calculator")

  calculator ! Multiplication(3, 1)
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)

  val handler = system.actorOf(Props[MessageHandler])
  system.eventStream.subscribe(handler,
                 classOf[UnhandledMessage]);
  calculator ! "Hello"
}


class MessageHandler extends Actor{
  def receive = {
    case UnhandledMessage(message, sender, recipient) =>
      println("unhandled message: "+message);
  }
}

/* Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
      Actor[akka://MySystem/user/calculator]
10 / 5 = 2
unhandled message: Hello
*/
```

Figure 1. Akka Example: Supervised Calculator

```scala
package sample.takka.SafeCalculator
import takka.actor.{ActorRef, ActorSystem, Props, Actor}

sealed trait Operation
case class Multiplication(m:Int, n:Int) extends Operation
case class Division(m:Int, n:Int) extends Operation

class Calculator extends Actor[Operation]{
  def typedReceive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m, n) =>
      println(m +" / "+ n +" = "+ (m/n))
  }
}
class SafeCalculator extends Actor[Operation] {
  override val supervisorStrategy =
    OneForOneStrategy() {
      case _: ArithmeticException =>
      println("ArithmeticException Raised to: "+typedSelf)
      Restart
    }
  val child:ActorRef[Operation] = typedContext.actorOf(
     Props[Operation, Calculator], "child")
  def typedReceive = {   case m => child ! m }
}
object SafeCalculatorTest extends App{
  val system = ActorSystem("MySystem")
  val calculator:ActorRef[Operation] =
       system.actorOf(Props[Operation, SafeCalculator],
                      "calculator")

  calculator ! Multiplication(3, 1)
  calculator ! Division(10, 0)
  calculator ! Division(10, 5)
  // calculator ! "Hello"
  //   compile error: type mismatch; found :
  //                  String("Hello") required:
  //   sample.takka.SupervisedCalculator.Operation

  System.out.println("Name server test")
  val calMul = system.actorFor[Multiplication]
                         ("akka://MySystem/user/calculator")
  calMul ! Multiplication(3, 2)
  Thread.sleep(1000)
  val calStr = system.actorFor[String]
                          ("akka://MySystem/user/calculator")
  // Exception raised before this line is reached
  calStr ! "Hello"
}
/* Terminal Output:
3 * 1 = 3
java.lang.ArithmeticException: / by zero
ArithmeticException Raised to:
      Actor[akka://MySystem/user/calculator]
10 / 5 = 2
Name server test
3 * 2 = 6
Exception in thread "main" java.lang.Exception:
      ActorRef[akka://MySystem/user/calculator] does not
      exist or does not have type ActorRef[String]
*/
```

Figure 2. TAkka Example: Supervised Calculator

```
package akka.actor

abstract class ActorRef
  def !(message: Any):Unit


trait  Actor
  def receive:PartialFunction[Any, Unit]
  val self: ActorRef
  private val contrext: ActorContext
  var supervisorStrategy: SupervisorStrategy


trait ActorContext
   def actorOf(props: Props): ActorRef

   def actorOf(props: Props, name: String): ActorRef

  def actorFor(path: String): ActorRef

  def setReceiveTimeout(timeout: Duration): Unit
  def become(behavior: PartialFunction[Any, Unit],
             discardOld:Boolean = true): Unit

  def unbecome(): Unit


final case class Props(deploy: Deploy,
    clazz: Class[_],
    args: immutable.Seq[Any])


object Props extends Serializable
  def apply(creator: =>Actor): Props
  def apply(actorClass: Class[_ <: Actor]): Props
  def apply[T <: Actor]()
      (implicit arg0: Manifest[T]): Props


abstract class SupervisorStrategy
case class OneForOneStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
case class OneForAllStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
```

Figure 3. Akka API

```
package takka.actor

abstract class ActorRef[-M](implicit mt:Manifest[M])
  def !(message: M):Unit
  def publishAs[SubM<:M]
      (implicit smt:Manifest[SubM]):ActorRef[SubM]


abstract class Actor[M:Manifest] extends akka.actor.Actor
  def typedReceive:M=>Unit
  val typedSelf:ActorRef[M]
  private val typedContext:ActorContext[M]
  var supervisorStrategy: SupervisorStrategy


abstract class ActorContext[M:Manifest]
  def actorOf [Msg] (props: Props[Msg])
             (implicit mt: Manifest[Msg]): ActorRef[Msg]
  def actorOf [Msg] (props: Props[Msg], name: String)
             (implicit mt:Manifest[Msg]): ActorRef[Msg]
  def actorFor [Msg] (path: String)
             (implicit mt:Manifest[Msg]):  ActorRef[Msg]
  def setReceiveTimeout(timeout: Duration): Unit
  def become[SupM >: M](behavior: SupM=>Unit)
           (implicit smt:Manifest[SupM]):ActorRef[SupM]

case class BehaviorUpdateException(smt:Manifest[_],
      mt:Manifest[_]) extends  Exception(smt + "must be a
      supertype of "+mt+".")


final case class Props[-T] (props: akka.actor.Props)


object Props extends Serializable
  def apply[T](creator: => Actor[T]): Props[T]
  def apply[T](actorClass: Class[_<: Actor[T]]):Props[T]
  def apply[T, A<:Actor[T]]
      (implicit arg0: Manifest[A]): Props[T]


abstract class SupervisorStrategy
case class OneForOneStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
case class OneForAllStrategy(restart:Int = -1,
            time:Duration = Duration.Inf)
            (decider: Throwable => Directive)
            extends SupervisorStrategy
```

Figure 4. TAkka API

For ease of use, `ActorRef` provides a `publishAs` method that casts an actor reference to a version that only accepts a subset of supported messages. The `publishAs` method encapsulates the process of type cast `ActorRef`, a contravariant type. We believe that using the notation of the `publishAs` method can be more intuitive than thinking about contravariance and subtyping relationship when publishing an actor reference as different types in a complex application. In addition, type conversion using `publishAs` is statically type checked. More importantly, with the `publishAs` method, users can give a supertype of an actor reference on demand, without defining new types and recompiling affected classes in the type hierarchy. The last advantage is important in Scala because a library developer may not have access to code written by others.

3.3 Props and Actor Context

The type `Props` denotes the properties of an actor. An instance of type `Props[M]` is used when creating an actor of type `Actor[M]`. Unlike an actor reference, which is the interface for receiving messages, an actor context describes the actor's view of the outside world. Because each actor defines an independent computation, an actor context is private to the corresponding actor. From its actor context, an actor can (i) retrieve an actor reference corresponding to a given actor path using the `actorFor` method, (ii) create a child actor with a system-generated or user-specified name using one of the `actorOf` methods, (iii) set a timeout denoting the time within which a new message must be received using the `setReceiveTimeout` method, and (iv) update its behaviours using the `become` method.

```scala
package sample.akka
import akka.actor.{ActorRef, ActorSystem, Props, Actor}



case class Multiplication(m:Int, n:Int)

case class Upgrade(advancedCalculator:
                   PartialFunction[Any,Unit])

class CalculatorServer extends Actor {
  def receive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Upgrade(advancedCalculator) =>
      println("Upgrading ...")
      context.become(advancedCalculator)
  }
}

object CalculatorUpgrade extends App {
  val system = ActorSystem("CalculatorSystem")
  val simpleCal:ActorRef =
      system.actorOf(Props[CalculatorServer],
      "calculator")

  simpleCal ! Multiplication(5, 1)

  case class Division(m:Int, n:Int)

  def advancedCalculator:PartialFunction[Any,Unit] = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
    case Upgrade(_) =>
      println("Upgraded.")
  }

   simpleCal ! Upgrade(advancedCalculator)
   val advancedCal = system.actorFor
      ("akka://CalculatorSystem/user/calculator")
   advancedCal ! Multiplication(5, 3)
   advancedCal ! Divison(10, 3)
   advancedCal ! Upgrade(advancedCalculator)
}
/*  Terminal Output:
5 * 1 = 5
Upgrading ...
5 * 3 = 15
10 / 3 = 3
Upgraded.
 */
```

Figure 5. Akka Example: Behaviour Upgrade

```scala
package sample.takka
import takka.actor.{ActorRef, ActorSystem, Props, Actor}

trait Operation
trait BasicOperation extends Operation
case class Multiplication(m:Int, n:Int)
     extends BasicOperation
case class Upgrade[Op >: BasicOperation]
     (advancedCalculator:Op=>Unit) extends
     BasicOperation
class CalculatorServer extends Actor[BasicOperation] {
  def typedReceive = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Upgrade(advancedCalculator) =>
      println("Upgrading ...")
      typedContext.become(advancedCalculator)
  }
}

object CalculatorUpgrade extends App {
  val system = ActorSystem("CalculatorSystem")
  val simpleCal:ActorRef[BasicOperation] =
       system.actorOf( Props[BasicOperation,
          CalculatorServer], "calculator")

  simpleCal ! Multiplication(5, 1)

  case class Division(m:Int, n:Int) extends Operation

  def advancedCalculator:Operation=>Unit = {
    case Multiplication(m:Int, n:Int) =>
      println(m +" * "+ n +" = "+ (m*n))
    case Division(m:Int, n:Int) =>
      println(m +" / "+ n +" = "+ (m/n))
    case Upgrade(_) =>
      println("Upgraded.")
  }

  simpleCal ! Upgrade(advancedCalculator)
  val advancedCal = system.actorFor[Operation]
     ("akka://CalculatorSystem/user/calculator")
  advancedCal ! Multiplication(5, 3)
  advancedCal ! Division(10, 3)
  advancedCal ! Upgrade(advancedCalculator)
}
/*  Terminal Output:
5 * 1 = 5
Upgrading ...
5 * 3 = 15
10 / 3 = 3
Upgraded.
 */
```

Figure 6. TAkka Example: Behaviour Upgrade

3.4 Reusing Akka Supervisor Strategies in TAkka

None of the supervisor strategies in Figure 4 require a type-parameterized class during construction. Therefore, from the perspective of API design, it is easy to reuse Akka supervisor strategies in TAkka. As actors communicate with each other by sending messages, system messages for supervision purposes should be handled by all actors. To keep the API simple, we separate the handler for system messages from the handler for other messages. In retrospect, the type parameter of the `Actor` class is not a supertype of system messages, whose types are private API in TAkka. Crucially, our design avoids the requirement for a union type, which is not provided by Scala.

3.5 Behaviour Upgrades

The `become` method enables behaviour upgrade of an actor. Figure 5 and 6 compare behaviour upgrade in Akka and TAkka. Unlike the Akka version, behaviour upgrade in TAkka *must* be backward compatible and *cannot* be rolled back. In other words, an actor must evolve into a version that is able to handle the original message patterns. The above decision is made so that a service published to users will not be unavailable later. Supporting behaviour upgrades in TAkka also requires that there is a suitable supertype defined in advance. This requirement is a weakness compared to Akka, which permits upgrading the behaviour to any syntactically correct implementation.

3.6 The Akka TypedActor class

Akka attempts to merge supervision and typed actors via a `TypedActor` class whose instance is initialised in a special way. A service of `TypedActor` object is invoked by method invocation instead of message passing. The Akka `TypedActor` class prevents some type errors but has two limitations. Firstly, TypedActor does not permit behaviour upgrade. Secondly, avoiding the type pollution problem (Section 6) by using Akka typed actors is the same cumbersome as using a simple object-oriented model, where supertypes need to be defined in advance. In Scala, introducing a supertype in a type hierarchy requires modification to all affected classes, whose source code may not be accessible by application developers.

4. Typed Name Server

An important part of distributed infrastructure is a name server, which maps names to a dynamically typed value. A name can be encoded as a `Symbol` in Scala so that names which represent the same string have the same value. As a value retrieved from a name server is *dynamically typed*, it needs to be checked and cast to the expected type at the client side before using it.

To overcome the limitations of the untyped name server, we design a typed name server. A typed name server maps each registered typed name to a value of the corresponding type, and allows look-up of a value by giving a typed name. A typed name, `TSymbol`, is a name shipped with a type descriptor. A typed value, `TValue`, is a value shipped with a type descriptor, which describes a super type of the most precise type of that value. `TSymbol` and `TValue` can be defined as in Figure 7. The APIs of a dynamic typed name server and a typed name server are given in Figure 8 and 9 respectively.

```
1 case class TSymbol[T:Manifest](val s:Symbol) {
2    private [takka] val t:Manifest[_] = manifest[T]
3    override def hashCode():Int = s.hashCode()
4    override def equals(that: Any) :Boolean = {
5      case ts: TSymbol[_] => ts.t.equals(this.t) &&
           ts.s.equals(this.s)
6      case _ => false
7    }
8 }
9 case class TValue[T:Manifest](val value:T){
10  private [takka] val t:Manifest[_] = manifest[T]
11 }
```

Figure 7. TSymbol and TValue

Each TAkka actor system contains a typed name server. The typed name server is used when the actor is created and when an actor reference is requested. When an actor is created, the actor records a map from a typed actor path and the typed actor reference for the created actor. Upon retrieving a typed actor reference, line 47 in Figure 2 for example, the typed name server checks if the typed actor path matches any record.

5. Evolution, Not Revolution

Akka systems can be smoothly migrated to TAkka systems. In other words, existing systems can evolve to introduce more types, rather than requiring a revolution where all actors and interactions must be typed. The above property is analogous to adding generics to Java programs. Java generics are carefully designed so that programs without generic types

```
1
2 object NameServer
3
4   def set(name:Symbol, value:Any):Boolean
5   def unset(name:Symbol):Boolean
6   def get(name:Symbol):Option[Any]
```

Figure 8. Dynamic Typed Name Server

```
1 package takka.nameserver
2 object NameServer
3   @throws(classOf[NamesHasBeenRegisteredException])
4   def set[T:Manifest](name:TSymbol[T], value:T):Boolean
5   def unset[T:Manifest](name:TSymbol[T]):Boolean
6   def get[T:Manifest](name:TSymbol[T]):Option[T]
7 case class
      NamesHasBeenRegisteredException(name:TSymbol[_])extends
      Exception("Name "+name+" has been registered.")
```

Figure 9. Static Typed Name Server

can be partially replaced by an equivalent generic version (evolution), rather than requiring generic types everywhere (revolution) (Naftalin and Wadler 2006).

Section 2 presents how to define and use a safe calculator in the Akka and TAkka systems respectively. Think of a `SafeCalculator` acotor as a service and its reference as a client interface. This section shows how to upgrade the Akka version to the TAkka version gradually, either upgrading the service implementation first or the client interface.

5.1 TAkka Service with Akka Client

It is often the case that an actor-based service is implemented by one organization but used in a client application implemented by another. Let us assume that a developer decides to upgrade the service using TAkka actors, for example, by upgrading the Socko Web Server (Imtarnasan and Bolton 2012), the Gatling stress testing tool (Excilys Group 2012), or the core library of Play (Typesafe Inc. (b) 2013), as we do in Section 7. Will the upgrade affect legacy client applications built using the Akka library? Fortunately, no changes are required at all.

As the TAkka Actor class inherits the Akka `Actor` class, it can be used to create an Akka actor. For example, the object `akkaCal`, created at line 5 in Figure 10, is created from a TAkka actor and used as an Akka actor reference. After the service developer has upgraded all actors to equivalent TAkka versions, the developer may want to start a TAkka actor system. Until that time, the developer can create TAkka actor references but publish their untyped version to users who are working in the Akka environment (line 19). As a result, no changes are required for a client application that uses Akka actor references. Because an Akka actor reference accepts messages of any type, messages of unexpected type may be sent to TAkka actors. As a result, handlers for the `UnhandledMessage` event is required in a careful design (line 10 and 20).

5.2 Akka Service with TAkka Client

Sometimes developers want to update the client code or API before upgrading the service implementation. For example, a developer may not have access to the service implementation; or the service implementation may be large, so the developer may want to upgrade the library gradually.

Users can initialize a TAkka actor reference by providing an Akka actor reference and a type parameter. In Figure 11, we re-use the Akka calculator, initialise it in an Akka actor

```
import sample.takka.SafeCalculator.SafeCalculator

object TSAC extends App {
  val akkasystem = akka.actor.ActorSystem("AkkaSystem")
  val akkaCal = akkasystem.actorOf(
    akka.actor.Props[SafeCalculator], "acal")
  val handler = akkasystem.actorOf(
    akka.actor.Props(new MessageHandler(akkasystem)))

  akkasystem.eventStream.subscribe(handler,
     classOf[UnhandledMessage]);
  akkaCal ! Multiplication(3, 1)
  akkaCal ! "Hello Akka"

  val takkasystem =
       takka.actor.ActorSystem("TAkkaSystem")
  val takkaCal = takkasystem.actorOf(
     takka.actor.Props[String, TAkkaStringActor], "tcal")

  val untypedCal= takkaCal.untypedRef
  takkasystem.system.eventStream.subscribe(
    handler,classOf[UnhandledMessage]);
  untypedCal ! Multiplication(3, 2)
  untypedCal ! "Hello TAkka"
}
/* Terminal output:
3 * 1 = 3
unhandled message:Hello Akka
3 * 2 = 6
unhandled message:Hello TAkka
*/
```

Figure 10. TAkka Service with Akka Client

system, and obtain an Akka actor reference. Then, we wrap the Akka actor reference as a TAkka actor reference, `takkaCal`, which only accepts messages of type `Operation`.

6. The Type Pollution Problem

In a system with multiple components, different components may require different interfaces; since all messages are received in the same mailbox, a naive approach would be to set the type to the union of all the interfaces, causing each component to see a type containing messages not intended for it—an issue we dub the Type Pollution Problem.

We illustrate the Type Pollution Problem and its solution on an instance of the Model-View-Controller pattern (Burbeck 1987). The Model and View have separate interfaces to the Controller, and neither should see the interface used by the other. However, the naive approach would have the Controller message type contain all the messages the Controller receives, from both the Model and the View. A similar problem can occur in a multi-tier architecture (Fowler 2002), where an intermediary layer interfaces with both the layer above and the layer below.

One solution to the type pollution problem is using separate channels for distinct parties. For instance, in Model-View-Controller, one channel would communicate between Model and Controller, and a distinct channel communicate between Model and View. Programming models that support this solution includes the join-calculus (Fournet and Gonthier 2000) and the typed π-calculus (Sangiorgi and Walker 2001). Can we gain similar advantages for a system based on actors rather than channels?

TAkka solves the type pollution problem with subtyping. The code outline in Figure 12 summarises a Tic-Tac-Toe example in the TAkka code repository (HE 2014), which

```
import sample.akka.SafeCalculator.SafeCalculator

object ASTC extends App {
  val system = akka.actor.ActorSystem("AkkaSystem")
  val akkaCal = system.actorOf(
      akka.actor.Props[SafeCalculator], "calculator")
  val takkaCal = new takka.actor.ActorRef[Operation]{
    val untypedRef = akkaCal
  }
  takkaCal ! Multiplication(3, 1)
// takkaCal ! "Hello"
// compile error: type mismatch;
//  found : String("Hello") required:
//    sample.takka.SupervisedCalculator.Operation
}
/* Terminal output:
3 * 1 = 3
*/
```

Figure 11. Akka Service with TAkka Client

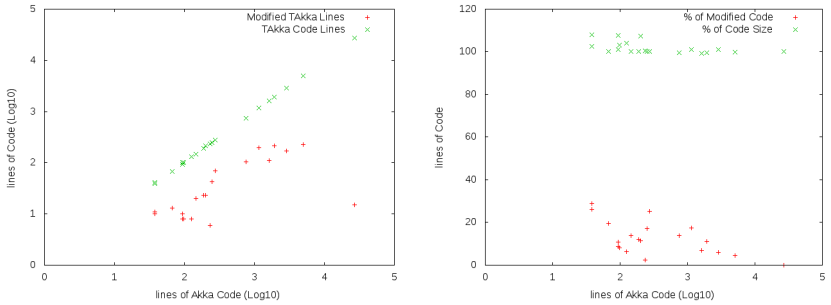
```
1  trait ControllerMessage
2  trait V2CMessage extends ControllerMessage
3  // sub-classes of V2CMessage messages go here
4  trait M2CMessage extends ControllerMessage
5  // sub-classes of M2CMessage messages go here
6  trait C2VMessage
7  case class ViewSetController
8    (controller:ActorRef[V2CMessage]) extends C2VMessage
9  trait C2MMessage
10 case class ModelSetController
11   (controller:ActorRef[M2CMessage]) extends C2MMessage
12
13 class View extends Actor[C2VMessage] {
14   private var controller:ActorRef[V2CMessage]
15   // rest of implementation
16 }
17 class Model extends Actor[C2MMessage] {
18   private var controller:ActorRef[M2CMessage]
19   // rest of implementation
20 }
21
22 class Controller(model:ActorRef[C2MMessage],
23      view:ActorRef[C2VMessage])
24      extends Actor[ControllerMessage] {
25   override def preStart() = {
26     model ! ModelSetController
27       (typedSelf.publishAs[M2CMessage])
28     view ! ViewSetController
29       (typedSelf.publishAs[V2CMessage])
30   }
31   // rest of implementation
32 }
```

Figure 12. Outline for Model-View-Controller

uses the Model-View-Controller pattern. Traits `V2CMessage` and `M2CMessage` represent the types of messages expected by the View and the Model respectively. Both are subtypes of `ControllerMsg`, which represents the type of all messages expected by the controller. In the code, the Controller actor publishes itself at different types to the View actor and the Model actor (lines 26–29), by sending appropriate initialisation messages. In line 27, `typedSelf` is of type `ActorRef[ControllerMsg]` while `ModelSetController` expects a parameter of type `ActorRef[M2CMessage]`. Since `ActorRef` is contravariant in its type parameter, the call is correct even if the call to `publishAs` is omitted; the call is to make the programmer's intent explicit and allows the compiler to catch more errors.

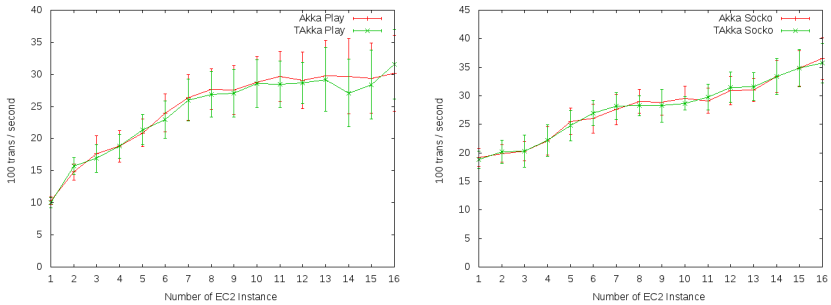
Source	Example	Akka Code Lines	Modified TAkka Lines	% of Modified Code	TAkka Code Lines	% of Code Size
Small Examples	String Processor	25	11	44	22	88
	Supervised Calculator	38	11	29	41	108
	Behaviour Upgrade	38	10	26	39	102
	NQueens	235	6	3	236	100
BenchErl Examples	bang	93	8	8.6	94	101
	big	93	10	11	100	108
	ehb	201	23	11	216	107
	mbrot	125	8	6	130	104
	ran	98	8	2.6	101	103
	serialmsg	146	20	14	146	100
Quviq (Arts et al. 2006)	ATM simulator	1148	199	17.3	1160	101
	Elevator Controller	2850	172	9.3	2878	101
Akka Documentation (Typesafe Inc. (a) 2012)	Ping Pong	67	13	19.4	67	100
	Dining Philosophers	189	23	12.1	189	100
	Distributed Calculator	250	43	17.2	250	100
	Fault Tolerance	274	69	25.2	274	100
Other Open Source Akka Applications	Barber Shop (Zachrison 2012)	754	104	13.7	751	99
	EnMAS (Doyle and Allen 2012)	1916	213	11.1	1909	100
	Socko Web Server (Imtarnasan and Bolton 2012)	5024	227	4.5	5017	100
	Gatling (Excilys Group 2012)	1635	111	6.8	1623	99
	Play Core (Typesafe Inc. (b) 2013)	27095	15	0.05	27095	100
geometric mean		354.1	30.2	8.5	360.1	101.7

Table 1. Results of Expressiveness Evaluation



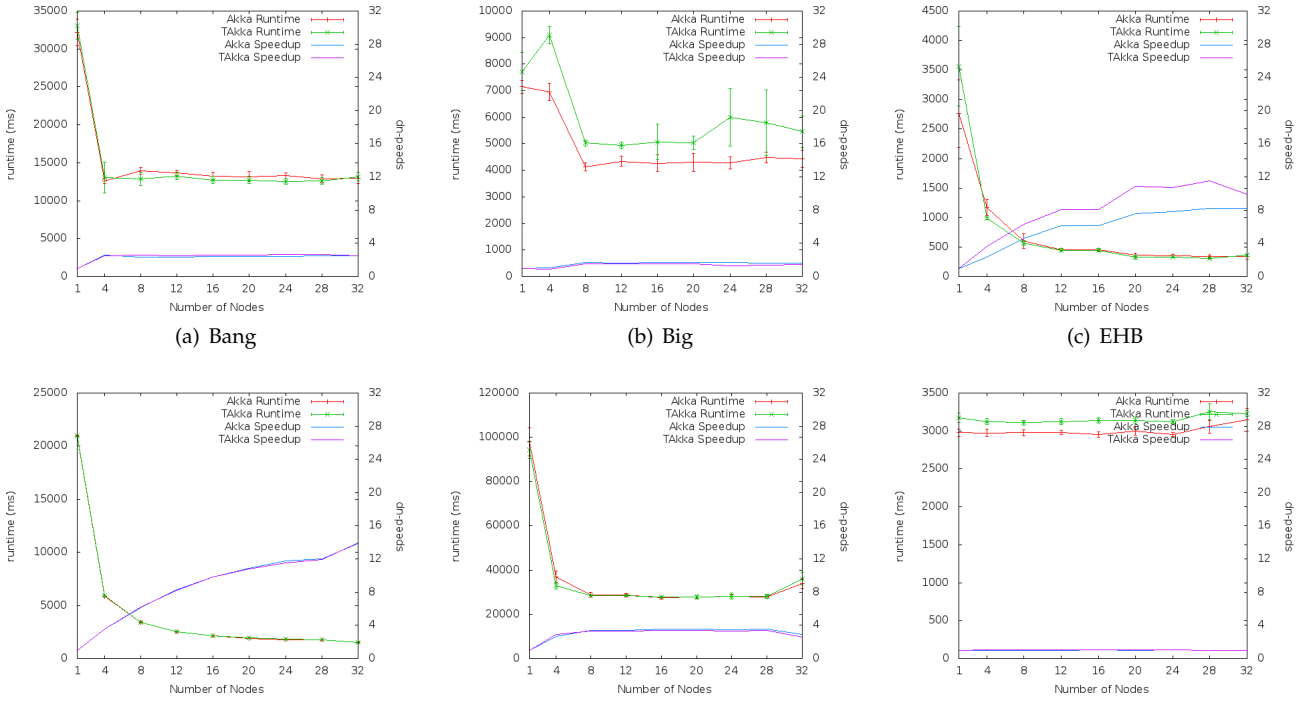
(a) Code Size: Absolute Lines (b) Code Size: Relative Lines

Figure 13. Code Size Evaluation



(a) Throughput: Play (b) Throughput: Socko

Figure 14. Throughput Benchmarks



(a) Bang (b) Big (c) EHB

(d) MBrot (e) RAN (f) SerialMsg

Figure 15. Runtime & Scalability Benchmarks

7. Expressiveness

This section investigates whether the type discipline enforced by TAkka restricts the expressibility of Akka. Table 1 lists the examples used for expressiveness checks. Examples are selected from Quviq (Arts et al. 2006) and open source Akka projects to ensure that the main requirements for actor programming are not unintentionally neglected. Examples from Quviq are re-implemented using both Akka and TAkka. Examples from Akka projects are re-implemented using TAkka. Following standard practice, we assess the overall code modification and code size by calculating the geometric mean of all examples (Hennessy and Patterson 2006). The evaluation results in Table 1 show that when porting an Akka program to TAkka, about 8.5% lines of code need to be modified including additional type declarations. Sometimes, the code size can be smaller because TAkka code does not need to handle unexpected messages. On average, the total program size of Akka and TAkka applications are almost the same. Figure 13 reports the same result in a Scatter chart.

A type error is reported by the compiler when porting the Socko example (Imtarnasan and Bolton 2012) from its Akka implementation to its equivalent TAkka implementation. Socko is a library for building event-driven web services. The Socko designer defines a `SockoEvent` class to be the supertype of all events. One subtype of `SockoEvent` is `HttpRequestEvent`, representing events generated when an HTTP request is received. The designer further implements subclasses of the `Method` class, whose `unapply` method is intended to have an output of type `Option[HttpRequestEvent]`. The Socko designer made a type error in the method declaration so that the `unapply` has output type `Option[SockoEvent]`. The type error is not exposed in test examples because those examples only test HTTP events. The design flaw is exposed when rewriting Socko using TAkka.

8. Throughput and Scalability

This section investigates whether managing type information in TAkka reduces performance. The TAkka library is built on Akka so that code for shared features can be re-used. The three main sources of overheads in the TAkka implementation are: (i) the cost of adding an additional operational layer on top of Akka code, (ii) the cost of constructing type descriptors, and (iii) the cost of transmitting type descriptors in distributed settings. We assess the effects of the above overheads in terms of throughput and scalability.

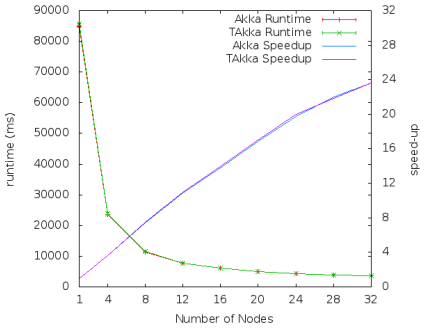
The example used in the throughput benchmark is the JSON serialization example (TechEmpower, Inc. 2013). The example was implemented using Akka Play, TAkka Play, Akka Socko, and TAkka Socko. All four versions of the web service are deployed to Amazon EC2 Micro instances (t1.micro), each of which has 0.615GB memory. The throughput is tested with up to 16 EC2 Micro instances. For each number of EC2 instances, 10 rounds of throughput measurement are executed to gather the average and standard deviation of the throughput. The results reported in Figure 14 show that web servers built using the Akka-based library and the TAkka-based library have very similar throughput.

We further investigate the speed-up of multi-node TAkka applications by porting 6 BenchErl benchmarks (Boudeville et al. 2012) which do not involve Erlang/OTP specific features. Each BenchErl benchmark spawns one master process and many child processes for a given task. Each child process performs a certain amount of computation and reports the result to the master process. The benchmarks are run on a 32 node Beowulf cluster at Heriot-Watt University. Each

Beowulf node comprises eight Intel 5506 cores running at 2.13GHz. All machines run under Linux CentOS 5.5. The Beowulf nodes are connected with a Baystack 5510-48T switch.

Figures 15 reports the results of the BenchErl benchmarks. We report the average and the standard deviation of the run-time of each example. Depending on the ratio of the computation time and the I/O time, benchmark examples scale at different levels. In all examples, TAkka and Akka implementations have almost identical run-time and scalability.

In the BenchErl examples, child processes are asked to execute the same computation a number of times. In contrast, distributed and cluster computing techniques are often used to solve a computationally expensive task by distributing sub-tasks to independent nodes. To simulate such a scenario, another benchmark, N-Queens Puzzle (Wikipedia 2014), is added. Finding all solutions of an N-Queen Puzzle is an NP-hard problem. Therefore, a suitable n makes the problem a good benchmark to demonstrate the advantage of cluster and distributed programming. Figure 16 reports the result when n is set to 14. The result shows that both the Akka and TAkka implementation have good scalability and similar efficiency.



(a) N-Queens Time

Figure 16. Benchmark: N-Queens Puzzle

9. Conclusion

The Akka library accepts dynamically typed messages. The TAkka library introduces a type-parameter for actor-related classes. The additional type-parameter specifies the communication interface of that actor. With the help of type-parameterized actors, unexpected messages to actors are rejected at compile time. We have shown that type-parameterized actors can form supervision trees in the same way as untyped actors (Section 3). We have shown that adding type parameter does not restrict expressiveness, and requires only small amounts of refactoring (Section 7). We have shown that TAkka does not introduced performance penalties (Section 8), with respect to throughput, efficiency, and scalability. The above results are encouraging for the use of types and supervision trees to implement reliable applications and improve the reliability of legacy applications with little effort.

Acknowledgments

The authors gratefully acknowledge the substantial help they have received from many colleagues who have shared their related results and ideas with us over the long period during which this paper was in preparation. Benedict Kavanagh and Danel Ahman for continuous comments and discussions. The RELEASE team for giving us access to the source code of the BenchErl benchmark examples. Thomas Arts from Quviq.com and Francesco Cesarini from Erlang Solutions for providing the Erlang source code of two examples used in their commercial training courses.

References

J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1. .

O. Boudeville, F. Cesarini, N. Chechina, K. Lundin, N. Papaspyrou, K. Sagonas, S. Thompson, P. Trinder, and U. Wiger. Release: a high-level paradigm for reliable large-scale server software. *Symposium on Trends in Functional Programming*, July 2012.

S. Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc), 1987. URL `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`.

C. Doyle and M. Allen. EnMAS: A new tool for multi-agent systems research and education. *Midwest Instruction and Computing Symposium*, 2012.

Excilys Group. Gatling: stress tool. http://gatling-tool.org/, 2012. Accessed on Oct 2012.

C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

J. HE. TAkka. `https://github.com/Jiansen/TAkka`, 2014. Accessed on May 2014.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 edition, Sept. 2006. ISBN 0123704901.

C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

V. Imtarnasan and D. Bolton. SOCKO Web Server. http://sockoweb.org/, 2012. Accessed on Oct 2012.

R. Kuhn and P. Vlugter. Parameterising Actor with Message type? `https://groups.google.com/forum/#!topic/akka-user/j-SgCS6JZoE`, 2011. Accessed on 17 Feb 2013.

R. Kuhn, J. Bonér, and P. Trinder. Typed akka actors. private communication, 2012.

M. Naftalin and P. Wadler. *Java Generics and Collections*, chapter Chapter 5: Evolution, Not revolution. O'Reilly Media, Inc., 2006. ISBN 0596527756.

D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521781779.

TechEmpower, Inc. Techempower web framework benchmarks. http://www.techempower.com/benchmarks/, 2013. Accessed on July 2013.

Typesafe Inc. (a). Akka Documentation: Release 2.0.2. http://doc.akka.io/docs/akka/2.0.2/Akka.pdf, 2012. Accessed on Oct 2012.

Typesafe Inc. (b). Play 2.2 documentation. http://www.playframework.com/documentation/2.2-SNAPSHOT/Home, 2013. Accessed on July 2013.

Wikipedia. Eight queens puzzle. `http://en.wikipedia.org/wiki/Eight_queens_puzzle`, 2014. [Online; accessed 30-March-2014].

M. Zachrison. Barbershop. https://github.com/cyberzac/BarberShop, 2012. Accessed on Oct 2012.