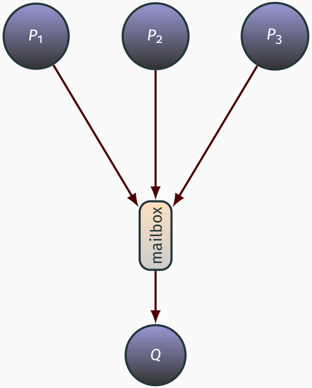
Mailbox Types for Unordered Interactions

Ugo de'Liguoro and Luca Padovani

University of Torino, Italy

Selective message processing



Context

- **many-to-one** communications
- **unpredictable** message order

Examples

- **actor model**
  Akka, Pony, Erlang, CAF, …
- **concurrent objects**
  locks, futures, semaphores, …

messages selected by tag, type, shape, …

From the Savina benchmark suite [Imam and Sarkar, 2014]

```
class Account(var balance: Double) extends ScalaActor[AnyRef] {
  override def process(msg: AnyRef) {
    msg match {
      case dm: DebitMessage =>
        balance += dm.amount
        sender.send(new ReplyMessage())
      case cm: CreditMessage =>
        balance -= cm.amount
        recipient.send(new DebitMessage(self, cm.amount))
        receive {
          case rm: ReplyMessage =>
            sender.send(new ReplyMessage())
        }
      case _: StopMessage => exit()
      case message =>
        val ex = new IllegalArgumentException("Unsupported␣message")
        ex.printStackTrace(System.err)
} } }
```

Goal

A type system for mailbox interactions

- well-typed processes interact **safely**
- don't receive **unexpected** messages
- don't leave **garbage** behind
- don't **deadlock**

Addressing "impure" actors to some extent

> *"We studied 15 large, mature, and actively maintained actor programs written in Scala and found that* ***80% of them mix the actor model with another concurrency model****"*
>
> *Tasharofi et al. [2013]*

From the Savina benchmark suite [Imam and Sarkar, 2014]

```
class Account(var balance: Double) extends AkkaActor[AnyRef] {
  override def process(msg: AnyRef) {
    msg match {
      case dm: DebitMessage =>
        balance += dm.amount
        sender() ! ReplyMessage
      case cm: CreditMessage =>
        balance -= cm.amount
        val recipient = cm.recipient.asInstanceOf[ActorRef]
        val future = ask(recipient, new DebitMessage(self,cm.amount))
        Await.result(future, Duration.Inf)
        sender() ! ReplyMessage
      case _: StopMessage => exit()
      case message =>
        val ex = new IllegalArgumentException("Unsupported message")
        ex.printStackTrace(System.err)
} } }
```

In this work

Mailbox Calculus

- **processes** using **first-class** mailboxes
- mixture of concurrency models (**actors**, **futures**, ...)

Mailbox Type System

- lack of message order is a **key feature** of mailbox types
- well-typed processes **interact safely** and **break even**

Syntax of the Mailbox Calculus

Asynchronous π-calculus + tagged messages + fail/free

Process

$$
\begin{array}{rcl}
P, Q & ::= & \mathtt{done} \\
& | & \mathtt{X}[\overline{u}] \\
& | & G \\
& | & u\,!\,\mathtt{m}[\overline{v}] \\
& | & P \mid Q \\
& | & (\nu a)P
\end{array}
$$

Guard

$$
\begin{array}{rcl}
G, H & ::= & \mathtt{fail}\ u \\
& | & \mathtt{free}\ u\,.\,P \\
& | & u\,?\,\mathtt{m}(\overline{x})\,.\,P \\
& | & G + H
\end{array}
$$

Mailbox Calculus

Mailbox = free-floating messages

$$\cdots u\,!\mathtt{A} \mid \cdots \mid u\,!\mathtt{B} \mid \cdots$$

Tags used to **select** messages from mailboxes

$$u\,!\mathtt{A} \mid u\,?\mathtt{A}\,.\,P + G \rightarrow P$$

Empty mailboxes are explicitly **deallocated**

$$(\nu u)(\mathtt{free}\; u\,.\,P + G) \rightarrow P$$

Processes may **fail**

$$(\nu u)(\cdots \mathtt{fail}\; u \cdots)$$

A simple example: the lock

$$\begin{aligned}\mathsf{Idle}(\mathit{lock}) &\triangleq \mathtt{free}\ \mathit{lock}.\mathtt{done} \\ &+ \mathit{lock}?\mathtt{acquire}(\mathit{user}).(\mathit{user}!\mathtt{reply}[\mathit{lock}] \mid \mathsf{Busy}[\mathit{lock}]) \\ &+ \mathit{lock}?\mathtt{release}.\mathtt{fail}\ \mathit{lock}\end{aligned}$$

$$\mathsf{Busy}(\mathit{lock}) \triangleq \mathit{lock}?\mathtt{release}.\mathsf{Idle}[\mathit{lock}]$$

- a lock is either idle or busy
- an idle lock **can** be acquired, but **cannot** be released
- a busy lock **must** be released

Properties

Definition
P is *mailbox conformant* if $P \not\to^* \mathcal{C}[\texttt{fail}\ a]$

Example (non-conformant process)

$$\text{Idle}(lock) \mid lock\,!\,\texttt{release}$$

Definition
P is *deadlock free* if $P \longrightarrow^* Q \not\to$ implies $Q \equiv \texttt{done}$

Example (conformant but deadlocking process)

$$\begin{array}{l} \text{Idle}(lock) \mid lock\,!\,\texttt{acquire}[user] \mid lock\,!\,\texttt{acquire}[user] \\ \mid user\,?\,\texttt{reply}(l_1)\,.\,user\,?\,\texttt{reply}(l_2)\,.\,(l_1\,!\,\texttt{release} \mid l_2\,!\,\texttt{release}) \end{array}$$

Syntax of Mailbox Types

$$\begin{array}{rrcl} \text{type} & \tau & ::= & \dagger E \\ \text{capability} & \dagger & ::= & ? \mid ! \\ \text{pattern} & E & ::= & \mathbb{0} \mid \mathbb{1} \mid \mathtt{m}[\overline{\tau}] \mid E + F \mid E \cdot F \mid E^* \end{array}$$

Capabilities

- ? = mailbox with **negative** balance ($\sim$ used for **inputs**)
- ! = mailbox with **positive** balance (used for **outputs**)

Patterns

- **commutative Kleene algebra** over message types $\mathtt{m}[\overline{\tau}]$

$$\dagger \mathtt{A} \cdot \mathtt{B} = \dagger \mathtt{B} \cdot \mathtt{A}$$

Lock's mailbox

$$?\mathtt{acquire}[!\mathtt{reply}[!\mathtt{release}]]^*$$

$$\begin{aligned}\mathsf{Idle}(\mathit{lock}) \triangleq\ & \mathtt{free}\ \mathit{lock}.\mathtt{done} \\ & + \mathit{lock}?\mathtt{acquire}(\mathit{user}).(\mathit{user}!\mathtt{reply}[\mathit{lock}] \mid \mathsf{Busy}[\mathit{lock}]) \\ & + \mathit{lock}?\mathtt{release}.\mathtt{fail}\ \mathit{lock}\end{aligned}$$

$$\mathsf{Busy}(\mathit{lock}) \triangleq \mathit{lock}?\mathtt{release}.\mathsf{Idle}[\mathit{lock}]$$

$$?\mathtt{release} \cdot \mathtt{acquire}[\cdots]^*$$

Typing Judgments

$$\Gamma \vdash P$$

Intuition

- $\Gamma =$ messages **produced** by $P -$ messages **consumed** by P

Consequence

- all types in Γ are $?\mathbb{1} \iff P$ **breaks even**

Example of typing derivation 1

$$\dfrac{\dfrac{}{u : \,!\mathsf{B} \vdash u!\mathsf{B}} \qquad \dfrac{\dfrac{}{u : \,!\mathsf{A} \vdash u!\mathsf{A}} \qquad \dfrac{\dfrac{\vdots}{u : ?\mathsf{B} \vdash P}}{u : ?\mathsf{A} \cdot \mathsf{B} \vdash u?\mathsf{A}.P}}{u : ?\mathsf{B} \vdash u!\mathsf{A} \mid u?\mathsf{A}.P}}{u : ?\mathbb{1} \vdash u!\mathsf{B} \mid u!\mathsf{A} \mid u?\mathsf{A}.P}$$

Example of typing derivation 2

$$
\dfrac{
  \dfrac{
    \dfrac{}{u : !\mathsf{B} \vdash u!\mathsf{B}}
    \qquad
    \dfrac{}{u : !\mathsf{A} \vdash u!\mathsf{A}}
  }{u : !\mathsf{B} \cdot \mathsf{A} \vdash u!\mathsf{B} \mid u!\mathsf{A}}
  \qquad
  \dfrac{
    \dfrac{\vdots}{u : ?\mathsf{B} \vdash P}
  }{u : ?\mathsf{A} \cdot \mathsf{B} \vdash u?\mathsf{A}.P}
}{u : ?\mathbb{1} \vdash u!\mathsf{B} \mid u!\mathsf{A} \mid u?\mathsf{A}.P}
$$

Example of typing derivation 2

$$
\dfrac{\dfrac{}{u : \,!\mathsf{B} \vdash u!\mathsf{B}} \qquad \dfrac{}{u : \,!\mathsf{A} \vdash u!\mathsf{A}}}{u : \,!\mathsf{A} \cdot \mathsf{B} \vdash u!\mathsf{B} \mid u!\mathsf{A}} \qquad \dfrac{\dfrac{\vdots}{u : \,?\mathsf{B} \vdash P}}{u : \,?\mathsf{A} \cdot \mathsf{B} \vdash u?\mathsf{A}.P}
$$
$$
\overline{\quad u : \,?\mathbb{1} \vdash u!\mathsf{B} \mid u!\mathsf{A} \mid u?\mathsf{A}.P \quad}
$$

Example: Typing a Lock

$$
\begin{aligned}
\mathsf{Idle}(lock) \triangleq\ & \mathtt{free}\ lock.\mathtt{done} \\
+\ & lock?\mathtt{acquire}(user).(user!\mathtt{reply}[lock] \mid \mathsf{Busy}[lock]) \\
+\ & lock?\mathtt{release}.\mathtt{fail}\ lock
\end{aligned}
$$

$?\mathtt{acquire}^*$

Example: Typing a Lock

$$\begin{aligned}\mathsf{Idle}(\mathit{lock}) \triangleq\ & \mathtt{free}\ \mathit{lock}.\mathtt{done} \\ +\ & \mathit{lock}?\mathtt{acquire}(\mathit{user}).(\mathit{user}!\mathtt{reply}[\mathit{lock}] \mid \mathsf{Busy}[\mathit{lock}]) \\ +\ & \mathit{lock}?\mathtt{release}.\mathtt{fail}\ \mathit{lock}\end{aligned}$$

$$?\mathtt{acquire}^* = ?\mathbb{1} + \mathtt{acquire} \cdot \mathtt{acquire}^* + \mathtt{release} \cdot \mathbb{0}$$

Properties of Well-Typed Processes

Theorem (conformance)
If $\Gamma \vdash P$, then P is mailbox conformant

This process is **mailbox conformant** but **deadlocks**

$$\begin{array}{l} \quad \mathsf{Idle}(lock) \mid lock\,!\,\mathtt{acquire}[user] \mid lock\,!\,\mathtt{acquire}[user] \\ \mid user?\mathtt{reply}(l_1)\,.\,user?\mathtt{reply}(l_2)\,.\,(l_1\,!\,\mathtt{release} \mid l_2\,!\,\mathtt{release}) \end{array}$$

On deadlocks and mailbox dependencies

Definition (mailbox dependency)
There is a **dependency** between mailboxes u and v if either

- v occurs in the continuation of a process blocked on u
- v occurs in a message stored in u

Strategy

1. collect **mailbox dependencies** in a graph φ

$$\Gamma \vdash P :: \varphi$$

2. make sure the graph has **no cycles**

Properties of well-typed processes, strengthened

Theorem (deadlock freedom)
If $\emptyset \vdash P :: \varphi$*, then* P *is deadlock free*

Theorem (fair termination)
If $\emptyset \vdash P :: \varphi$ *for* P *finitely unfolding, then* $P \rightarrow^* Q$ *implies* $Q \rightarrow^*$ done

Corollary (garbage freedom)
Closed, well-typed, finitely-unfolding processes ***leave no garbage***

Artifact (not evaluated)

Mailbox Calculus Checker available from my home page

Main issues

- subtyping can be as complex as validity of Presburger formulas
- potentially **lots** of type annotations, **Newtonian program analysis** to the rescue [Esparza et al., 2010]

Final remarks

Summary

- mailbox calculus $\sim$ actors with **first-class/multiple** mailboxes
- mailbox types $\sim$ descriptions of **unordered** mailboxes

In the paper

- more examples (actors using **futures**, master-workers)
- encoding of **binary sessions** with **joins** and **forks**

Future work

- analyse real-world actor languages and libraries
- investigate analogies with **linear logic**

References

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. Newtonian program analysis. *J. ACM*, 57(6):33:1–33:47, November 2010. ISSN 0004-5411.

Shams Mahmood Imam and Vivek Sarkar. Savina - an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of AGERE! 2014*, pages 67–80. ACM, 2014.

Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In *Proceedings of ECOOP'13*, LNCS 7920, pages 302–326. Springer, 2013.