A Formal Model for Direct-style Asynchronous Observables

Philipp Haller
KTH Royal Institute of Technology, Sweden

Heather Miller
EPFL, Switzerland

27th Nordic Workshop on Programming Theory (NWPT)
Reykjavik University, Iceland, 21-23 October 2015
Context: Asynchronous Programming

- Thread-based, blocking abstractions
  - Direct-style programming model (easy of use), good debugging support
  - Not efficient, not scalable
- Event-based, non-blocking abstractions
  - Efficient, scalable
  - Hard to use: inversion of control, “callback hell”
  - Debugging support lacking
Background: Async Model

- A recent proposal for simplifying asynchronous programming
- Essence of the Async Model:
  1. A way to spawn an asynchronous computation \((\text{async})\), returning a (first-class) future
  2. A way to suspend an asynchronous computation \((\text{await})\) until a future is completed
- Result: a \textit{direct-style API for non-blocking futures}
- Practical relevance: F#, C# 5.0, Scala 2.11
Example

• Setting: Play Web Framework

• Task: Given two web service requests, when both are completed, return response that combines both results:

```scala
val futureDOY: Future[Response] = 
    WS.url("http://api.day-of-year/today").get
val futureDaysLeft: Future[Response] = 
    WS.url("http://api.days-left/today").get
```
Example

Using Scala Async

val respFut = async {
  val dayOfYear = await(futureDOY).body
  val daysLeft = await(futureDaysLeft).body
  Ok("" + dayOfYear + ": " + daysLeft + " days left!")
}

Example

Using plain Scala futures

```
futureDOY.flatMap { doyResponse =>
    val dayOfYear = doyResponse.body
    futureDaysLeft.map { daysLeftResponse =>
        val daysLeft = daysLeftResponse.body
        Ok("" + dayOfYear + ": " + daysLeft + " days left!")
    }
}
```

Using Scala Async

```
val respFut = async {
    val dayOfYear = await(futureDOY).body
    val daysLeft = await(futureDaysLeft).body
    Ok("" + dayOfYear + ": " + daysLeft + " days left!")
}
```
Problem

• Async model only supports futures
• What about streams of asynchronous events?
Asynchronous Streams
Asynchronous Streams

- Asynchronous event streams and push notifications: a fundamental abstraction for web and mobile apps
Asynchronous Streams

- Asynchronous event streams and push notifications: a fundamental abstraction for web and mobile apps
- Requirement: *extreme scalability and efficiency*
  - Precludes future-per-event implementations
  - Examples: Netflix, Samsung SAMI, ...
Asynchronous Streams

- Asynchronous event streams and push notifications: a fundamental abstraction for web and mobile apps
- Requirement: *extreme scalability and efficiency*
  - Precludes future-per-event implementations
  - Examples: Netflix, Samsung SAMI, ...
- Popular programming model: Reactive Extensions
  - Based on the duality of iterators and observers
Reactive Extensions: Essence

trait Observable[T] {
  def subscribe(obs: Observer[T]): Closable
}

trait Observer[T] {
  def onNext(v: T): Unit
  def onFailure(t: Throwable): Unit
  def onDone(): Unit
}

Observer\([T]\): Interactions

Erik Meijer: Your mouse is a database. CACM ’12
The Real Power: Combinators

Figure 7. The SelectMany operator.
Combinators: Example

def textChanges(tf: JTextField):
    Observable[String]

textChanges(textField)
    .flatMap(word => completions(word))
    .subscribe(observeChanges(output))
Problem

- Programming with reactive streams suffers from an inversion of control
- Requires explicit programming in continuation-passing style (CPS)
- Writing stateful combinators is difficult
RAY: Idea

- Unify Reactive Extensions and Async
- Introduce variant of `async {}` to create observables instead of futures: `rasync {}`
- Within `rasync {}`: enable *awaiting events of observables in direct-style*
- Create observables by yielding events from within `rasync {}`
RAY: Primitives

- `rasync[T] { }` - create `Observable[T]`
- `awaitNextOrDone(obs)` - awaits and returns `Some(next event of obs)`, or else returns `None` if `obs` has terminated
- `yieldNext(evt)` - yields next event of current observable
RAY: Simple Example

val forwarder = rasync[Int] {
  var next: Option[Int] = awaitNextOrDone(stream)
  while (next.nonEmpty) {
    yieldNext(next)
    next = awaitNextOrDone(stream)
  }
}
Formalization

Object-based calculus

\[
\begin{align*}
p & ::= \overline{cd} \; e \\
\overline{cd} & ::= \text{class } C \; \{ \overline{fd} \; \overline{md} \} \\
\overline{fd} & ::= \text{var } f : \sigma \\
\overline{md} & ::= \text{def } m(\overline{x} : \sigma) : \tau = e \\
\sigma, \tau & ::= \\
& \quad | \gamma \\
& \quad | \rho \\
\gamma & ::= \\
& \quad | \text{Boolean} \\
& \quad | \text{Int} \\
\rho & ::= \\
& \quad | C \\
& \quad | \text{Observable}[\sigma]
\end{align*}
\]
Expressions

\[ e ::= \]

- boolean
- integer
- variable
- null
- condition
- while loop
- selection
- assignment
- invocation
- instance creation
- let binding
- observable creation
- await next event
- yield event
Expressions

\[ e ::= \]

- \( b \)
- \( i \)
- \( x \)
- \( \text{null} \)
- \( \text{if} \ (x) \ {e} \ \text{else} \ {e'} \)
- \( \text{while} \ (x) \ {e} \)
- \( x.f \)
- \( x.f = y \)
- \( x.m(\overline{y}) \)
- \( \text{new} \ C(\overline{y}) \)
- \( \text{let} \ x = e \ \text{in} \ e' \)
- \( \text{rasync}[\sigma](\overline{y}) \ {e} \)
- \( \text{await}(x) \)
- \( \text{yield}(x) \)

expressions

- boolean
- integer
- variable
- null
- condition
- while loop
- selection
- assignment
- invocation
- instance creation
- let binding
- observable creation
- await next event
- yield event
Operational Semantics

- Small-step operational semantics
- Transitions for frames, frame stacks, and processes (sets of frame stacks)

\[
H(L(y)) = \langle \rho, FM \rangle \\
H, \langle L, \text{let } x = y.f \ in \ e \rangle^l \rightarrow H, \langle L[x \mapsto FM(f)], e \rangle^l
\]

(E-FIELD)

\[
\text{fields}(C) = \bar{f} \quad o \notin \text{dom}(H) \\
H' = H[o \mapsto \langle C, \bar{f} \mapsto L(\bar{y}) \rangle] \\
H, \langle L, \text{let } x = \text{new } C(\bar{y}) \ in \ e \rangle^l \rightarrow H', \langle L[x \mapsto o], e \rangle^l
\]

(E-NEW)
Reducing Frame Stacks

\[ H(L(y)) = \langle \rho, FM \rangle \quad \text{mbody}(\rho, m) = (\bar{x}) \rightarrow e' \]

\[ L' = [\bar{x} \mapsto L(\bar{z}), \text{this} \mapsto L(y)] \]

\[ H, \langle L, \text{let } x = y.m(\bar{z}) \text{ in } e \rangle^l \circ FS \rightarrow H, \langle L', e' \rangle^s \circ \langle L, e \rangle^l_x \circ FS \]

(E-METHOD)

\[ H, \langle L, y \rangle^s \circ \langle L', e \rangle^l_x \circ FS \rightarrow H, \langle L'[x \mapsto L(y)], e \rangle^l \circ FS \]

(E-RETURN)

\[ H, F \rightarrow H', F' \]

\[ H, F \circ FS \rightarrow H', F' \circ FS \]

(E-FRAME)
Observables

• A special kind of object

• State of an observable: running or done

\[ H(o) = \langle \text{Observable} [\sigma], \text{running}(\bar{F}, \bar{S}) \rangle \]

• Initial state: \( \text{running}(\epsilon, \epsilon) \)

• Running state: \( \text{running}(\bar{F}, \bar{S}) \)

• Terminated state: \( \text{done}(\bar{S}) \)
Waiters

\[ H(o) = \langle \text{Observable}[\sigma], \text{running}(\bar{F}, \bar{S}) \rangle \]
Waiters

\[ H(o) = \langle Observable[\sigma], \text{running}(\bar{F}, \bar{S}) \rangle \]

- **Waiters**: asynchronous frames of suspended observables
- Example of a waiter:

\[ F = \langle L, \text{let } x = \text{await}(y) \text{ in } t \rangle^{a(o, \bar{p})} \]
Heap Evolution

Heap Evolution property formalizes **permitted observable protocol state transitions**

**Definition 1** (Heap Evolution). Heap $H$ evolves to $H'$ wrt a set of observable ids $B$, written $H \leq_B H'$ if

(i) $\forall o \in \text{dom}(H').$ if $o \notin \text{dom}(H)$ and $H'(o) = \langle \psi, \text{running}(\bar{F}, \bar{S}) \rangle$ then $\bar{F} = \bar{S} = \varepsilon$, and

(ii) $\forall o \in \text{dom}(H)$.

- if $H(o) = \langle C, FM \rangle$ then $H'(o) = \langle C, FM' \rangle$,
- if $H(o) = \langle \psi, \text{done}(\bar{S}) \rangle$ then $H'(o) = \langle \psi, \text{done}(\bar{R} \cup \{(o', q')\}) \rangle$ where $\bar{S} = \bar{R} \cup \{(o', q)\}$, and
- if $H(o) = \langle \psi, \text{running}(\bar{F}, \bar{S}) \rangle$ then $H'(o) = \langle \psi, \text{running}(\bar{F}, \bar{S} \cup \{(o, [])\}) \rangle$ or $(H'(o) = \langle \psi, \text{running}(e, \bar{R}) \rangle$ and $\text{dom}(\bar{S}) = \text{dom}(\bar{R})$ or $(H'(o) = \langle \psi, \text{running}(\bar{F} \cup \bar{G}, \bar{S}) \rangle$, $\text{obsIds}(\bar{F}) \neq \text{obsIds}(\bar{G})$ and $\text{obsIds}(\bar{G}) \subseteq B$) or $H'(o) = \langle \psi, \text{done}(\bar{S}) \rangle$. 

Example: Terminating an Observable

\[ H(o) = \langle \text{Observable}[\sigma], \text{running}(\bar{F}, \bar{S}) \rangle \]
\[ \bar{R} = \text{resume}(\bar{F}, \text{None}) \quad Q = \{ R \circ \epsilon \mid R \in \bar{R} \} \]
\[ H_0 = H[o \mapsto \langle \text{Observable}[\sigma], \text{done}(\bar{S}) \rangle] \]
\[ \forall i \in 1 \ldots n. \quad H_i = H_{i-1}[p_i \mapsto \text{unsub}(o,p_i,H)] \]
\[ H, \{ \langle L, x \rangle^{a(o,\bar{p})} \circ FS \} \cup P \leadsto H_n, \{ FS \} \cup P \cup Q \tag{E-RAasync-Return} \]
Example: Terminating an Observable

\[
H(o) = \langle Observable[\sigma], \text{running}(\bar{F}, \bar{S}) \rangle
\]

\[
\bar{R} = \text{resume}(\bar{F}, \text{None}) \quad Q = \{ R \circ \epsilon \mid R \in \bar{R} \}
\]

\[
H_0 = H[o \mapsto \langle Observable[\sigma], \text{done}(\bar{S}) \rangle]
\]

\[
\forall i \in 1 \ldots n. \quad H_i = H_{i-1}[p_i \mapsto \text{unsub}(o, p_i, H)]
\]

\[
\frac{H, \{ \langle L, x \rangle ^{a(o, \bar{p})} \circ FS \} \cup P \leadsto H_n, \{ FS \} \cup P \cup Q}{(E-RAasync-Return)}
\]
Preservation of Heap Evolution

- Proving that reduction preserves heap evolution requires preserving non-interference properties
- Example: a given observable $o$ can only be waiting for exactly one other observable
- Requires observable ids of waiters to be distinct
Subject Reduction

Subject reduction theorem

- ensures *observable protocol*
- through *heap evolution* and *non-interference*

**Theorem 1** (Subject Reduction). If $\vdash H : \ast$ and $\vdash H \texttt{ ok}$ then:

1. If $H \vdash F : \sigma$, $H \vdash F \texttt{ ok}$ and $H, F \rightarrow H', F'$ then $\vdash H' : \ast, H' \vdash \texttt{ ok}$, $H' \vdash F' : \sigma$, $H' \vdash F' \texttt{ ok}$, and $\forall B. H \leq_B H'$.

2. If $H \vdash FS : \sigma$, $H \vdash FS \texttt{ ok}$ and $H, FS \rightarrow H', FS'$ then $\vdash H' : \ast, H' \vdash \texttt{ ok}$, $H' \vdash FS' : \sigma$, $H' \vdash FS' \texttt{ ok}$ and $H \leq_{obsIds(FS)} H'$.

3. If $H \vdash P : \ast$, $H \vdash P \texttt{ ok}$ and $H, P \rightsquigarrow H', P'$ then $\vdash H' : \ast, H' \vdash \texttt{ ok}$, $H' \vdash P' : \ast$ and $H' \vdash P' \texttt{ ok}$.
Subject Reduction

Subject reduction theorem

- ensures **observable protocol**
  - through **heap evolution** and **non-interference**

**Theorem 1** (Subject Reduction). If $H \vdash *$ and $H \vdash ok$ then:

1. If $H \vdash F : \sigma$, $H \vdash F \mathbin{ok}$ and $H, F \rightarrow H', F'$ then $H' \vdash *$, $H' \vdash ok$, $H' \vdash F' : \sigma$, $H' \vdash F' \mathbin{ok}$, and $\forall B. H \leq_B H'$.

2. If $H \vdash FS : \sigma$, $H \vdash FS \mathbin{ok}$ and $H, FS \rightarrow H', FS'$ then $H' \vdash *$, $H' \vdash ok$, $H' \vdash FS' : \sigma$, $H' \vdash FS' \mathbin{ok}$ and $H \leq_{\text{obsIds(FS)}} H'$.

3. If $H \vdash P : *$, $H \vdash P \mathbin{ok}$ and $H, P \rightarrow H', P'$ then $H' \vdash *$, $H' \vdash ok$, $H' \vdash P' : *$ and $H' \vdash P' \mathbin{ok}$. 
Subject Reduction

Subject reduction theorem

- ensures *observable protocol*
- through *heap evolution* and *non-interference*

**Theorem 1** (Subject Reduction). If \( \vdash H : \star \) and \( \vdash H \ ok \) then:

1. If \( \vdash H : F : \sigma \), \( \vdash H \ ok \) and \( H, F \rightarrow H', F' \) then \( \vdash H' : \star \), \( \vdash H' \ ok \), \( H' \vdash F' : \sigma \), and \( \forall B. H \leq_B H' \).

2. If \( \vdash H : FS : \sigma \), \( \vdash FS \ ok \) and \( H, FS \rightarrow H', FS' \) then \( \vdash H' : \star \), \( \vdash H' \ ok \), \( H' \vdash FS' : \sigma \), and \( H \leq_{obsIds(FS)} H' \).

3. If \( \vdash H : P : \star \), \( \vdash P \ ok \) and \( H, P \leadsto H', P' \) then \( \vdash H' : \star \), \( \vdash H' \ ok \), \( H' \vdash P' : \star \) and \( H' \vdash P' \ ok \).
Selected Related Work

- Bierman et al. *Pause 'n' Play: Formalizing Asynchronous C#*. ECOOP 2012
- Meijer. *Your mouse is a database*. CACM 55.5, 2012
Results

- RAY: unifies Async model and Reactive Extensions
- Operational semantics and static type system
- Proof of subject reduction
  - Based on non-interference properties
  - Ensures observable protocol
- Companion technical report provides details
- See http://www.csc.kth.se/~phaller/nwpt2015/
Results

- RAY: unifies Async model and Reactive Extensions
- Operational semantics and static type system
- Proof of subject reduction
  - Based on non-interference properties
  - Ensures observable protocol
- Companion technical report provides details
- See http://www.csc.kth.se/~phaller/nwpt2015/