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

KTH VETENSKAP OCH KONST

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

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 (*async*), returning a (first-class) future
  2. A way to suspend an asynchronous computation (*await*) until a future is completed
- Result: a *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:

```
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

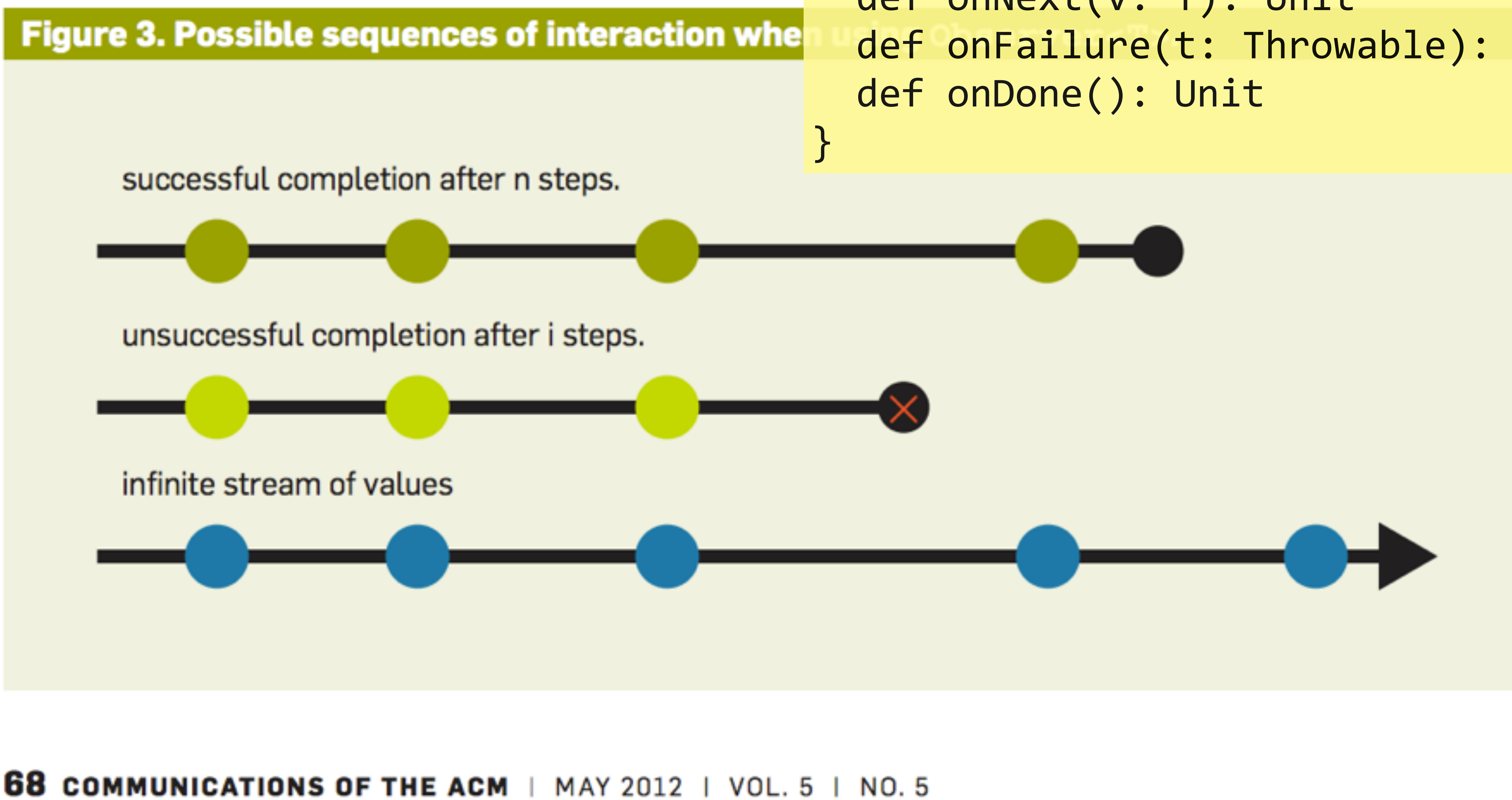
Reactive Extensions: Essence

```
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

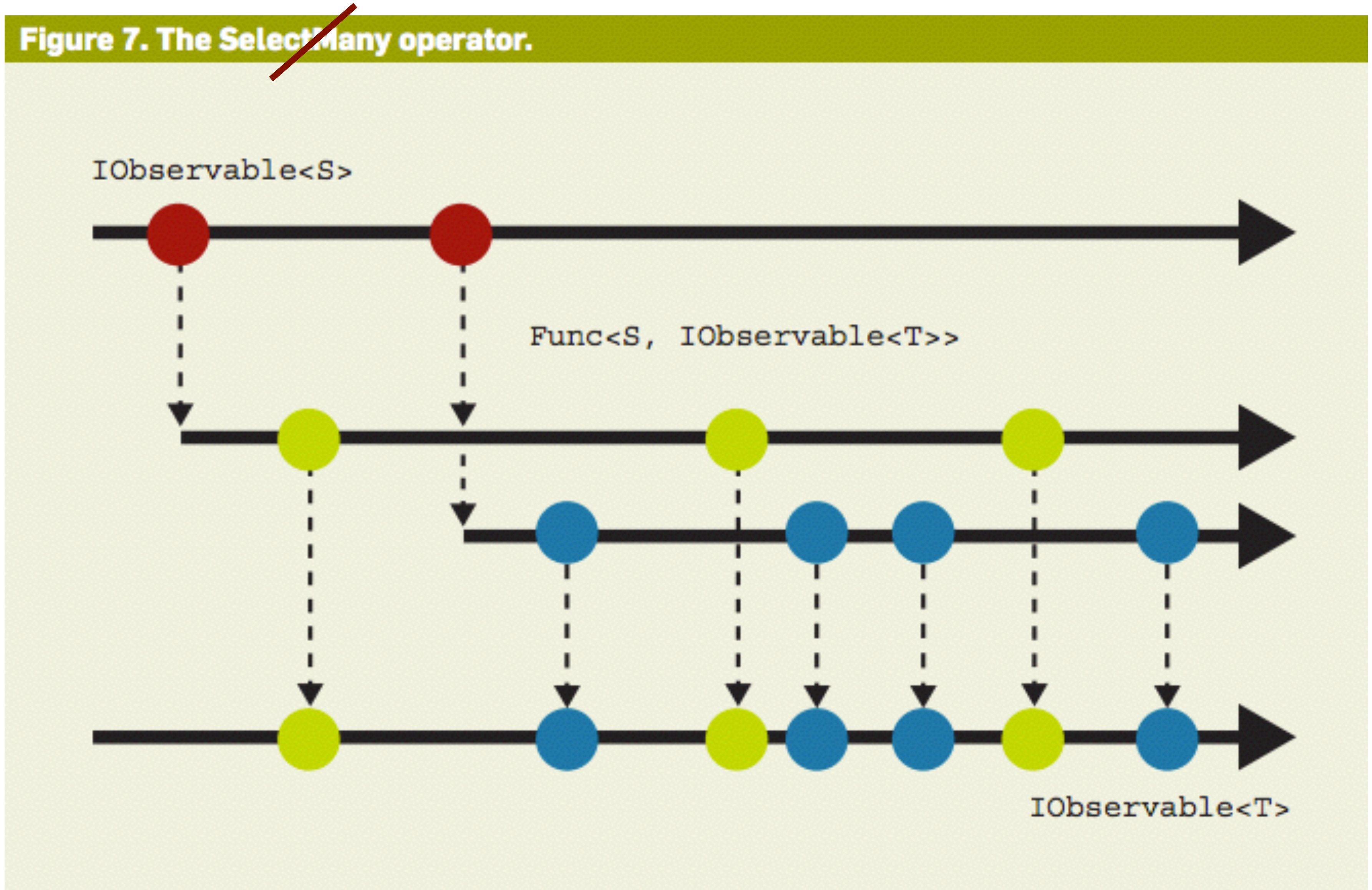
```
trait Observer[T] {
  def onNext(v: T): Unit
  def onFailure(t: Throwable): Unit
  def onDone(): Unit
}
```



Erik Meijer: Your mouse is a database. CACM ’12

The Real Power: Combinators

flatMap



Combinators: Example

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

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

$p ::= \overline{cd}\ e$	program
$cd ::= \texttt{class}\ C\ \{\overline{fd}\ \overline{md}\}$	class declaration
$fd ::= \texttt{var}\ f : \sigma$	field declaration
$md ::= \texttt{def}\ m(\overline{x : \sigma}) : \tau = e$	method declaration
$\sigma, \tau ::=$	type
$\mid \gamma$	value type
$\mid \rho$	reference type
$\gamma ::=$	value type
$\mid \texttt{Boolean}$	boolean
$\mid \texttt{Int}$	integer
$\rho ::=$	reference type
$\mid C$	class type
$\mid \texttt{Observable}[\sigma]$	observable type

Expressions

$e ::=$	expressions
$\mid \underline{b}$	boolean
$\mid \underline{i}$	integer
$\mid x$	variable
$\mid \texttt{null}$	null
$\mid \texttt{if}\ (x)\ \{e\}\ \texttt{else}\ \{e'\}$	condition
$\mid \texttt{while}\ (x)\ \{e\}$	while loop
$\mid x.f$	selection
$\mid x.f = y$	assignment
$\mid x.m(\overline{y})$	invocation
$\mid \texttt{new C}(\overline{y})$	instance creation
$\mid \texttt{let}\ x = e\ \texttt{in}\ e'$	let binding
$\mid \texttt{rasync[}\sigma\texttt{]}(\overline{y})\ \{e\}$	observable creation
$\mid \texttt{await}(x)$	await next event
$\mid \texttt{yield}(x)$	yield event

Expressions

$e ::=$		expressions
	$\mid \underline{b}$	boolean
	$\mid \underline{i}$	integer
	$\mid x$	variable
	$\mid \texttt{null}$	null
	$\mid \texttt{if}\ (x)\ \{e\}\ \texttt{else}\ \{e'\}$	condition
	$\mid \texttt{while}\ (x)\ \{e\}$	while loop
	$\mid x.f$	selection
	$\mid x.f = y$	assignment
	$\mid x.m(\overline{y})$	invocation
	$\mid \texttt{new C}(\overline{y})$	instance creation
	$\mid \texttt{let}\ x = e\ \texttt{in}\ e'$	let binding
	$\mid \texttt{rasync}[\sigma](\bar{y})\ \{e\}$	observable creation
	$\mid \texttt{await}(x)$	await next event
	$\mid \texttt{yield}(x)$	yield event

Operational Semantics

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

$$\frac{H(L(y)) = \langle \rho, FM \rangle}{H, \langle L, \mathtt{let}\ x = y.f\ \mathtt{in}\ e \rangle^l \longrightarrow H, \langle L[x \mapsto FM(f)], e \rangle^l} \quad \text{(E-FIELD)}$$

$$\frac{fields(C) = \bar{f} \qquad o \notin dom(H) \qquad H' = H[o \mapsto \langle C, \bar{f} \mapsto L(\bar{y}) \rangle]}{H, \langle L, \mathtt{let}\ x = \mathtt{new}\ C(\bar{y})\ \mathtt{in}\ e \rangle^l \longrightarrow H', \langle L[x \mapsto o], e \rangle^l} \quad \text{(E-NEW)}$$

Reducing Frame Stacks

$$\frac{H(L(y)) = \langle \rho, FM \rangle \qquad mbody(\rho, m) = (\overline{x}) \rightarrow e' \qquad L' = [\bar{x} \mapsto L(\bar{z}), \mathtt{this} \mapsto L(y)]}{H, \langle L, \mathtt{let}\ x = y.m(\bar{z})\ \mathtt{in}\ e \rangle^l \circ FS \ \twoheadrightarrow\ H, \langle L', e' \rangle^s \circ \langle L, e \rangle^l{}_x \circ FS} \quad \text{(E-METHOD)}$$

$$\frac{}{H, \langle L, y \rangle^s \circ \langle L', e \rangle^l{}_x \circ FS \ \twoheadrightarrow\ H, \langle L'[x \mapsto L(y)], e \rangle^l \circ FS} \quad \text{(E-RETURN)}$$

$$\frac{H, F \ \longrightarrow\ H', F'}{H, F \circ FS \ \twoheadrightarrow\ H', F' \circ FS} \quad \text{(E-FRAME)}$$

Observables

- A special kind of object
- State of an observable: running or done

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

- Initial state: $running(\epsilon, \epsilon)$
- Running state: $running(\bar{F}, \bar{S})$
- Terminated state: $done(\bar{S})$

Waiters

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

Waiters

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

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

$$F = \langle L, \texttt{let } x = \texttt{ await}(y) \texttt{ in } t \rangle^{a(o,\bar{p})}$$

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

(ii) $\forall o \in dom(H)$.

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

Example: Terminating an Observable

$$\frac{\begin{array}{c} H(o) = \langle \texttt{Observable}[\sigma], running(\bar{F}, \bar{S}) \rangle \\ \bar{R} = resume(\bar{F}, \texttt{None}) \qquad Q = \{R \circ \epsilon \mid R \in \bar{R}\} \\ H_0 = H[o \mapsto \langle \texttt{Observable}[\sigma], done(\bar{S}) \rangle] \\ \forall i \in 1 \ldots n.\ H_i = H_{i-1}[p_i \mapsto unsub(o, p_i, H)] \end{array}}{H, \{\langle L, x \rangle^{a(o,\bar{p})} \circ FS\} \cup P \ \rightsquigarrow\ H_n, \{FS\} \cup P \cup Q} \ (\text{E-RAsync-Return})$$

Example: Terminating an Observable

$$\frac{\begin{array}{c} H(o) = \langle \texttt{Observable}[\sigma], running(\bar{F}, \bar{S})\rangle \\ \bar{R} = resume(\bar{F}, \texttt{None}) \qquad Q = \{R \circ \epsilon \mid R \in \bar{R}\} \\ H_0 = H[o \mapsto \langle \texttt{Observable}[\sigma], done(\bar{S})\rangle] \\ \forall i \in 1 \ldots n.\ H_i = H_{i-1}[p_i \mapsto unsub(o, p_i, H)] \end{array}}{H, \{\langle L, x\rangle^{a(o,\bar{p})} \circ FS\} \cup P \rightsquigarrow H_n, \{FS\} \cup P \cup Q} \quad (\text{E-RAsync-Return})$$

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* $\vdash H : \star$ *and* $\vdash H$ **ok** *then:*

1. *If* $H \vdash F : \sigma$, $H \vdash F$ **ok** *and* $H, F \longrightarrow H', F'$ *then* $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash F' : \sigma$, $H' \vdash F'$ **ok**, *and* $\forall B.\ H \leq_B H'$.
2. *If* $H \vdash FS : \sigma$, $H \vdash FS$ **ok** *and* $H, FS \twoheadrightarrow H', FS'$ *then* $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash FS' : \sigma$, $H' \vdash FS'$ **ok** *and* $H \leq_{obsIds(FS)} H'$.
3. *If* $H \vdash P : \star$, $H \vdash P$ **ok** *and* $H, P \rightsquigarrow H', P'$ *then* $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash P' : \star$ *and* $H' \vdash P'$ **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* $H \vdash F : \sigma$, $H \vdash F$ **ok** *and* $H, F \longrightarrow H', F'$ *then* $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash F' : \sigma$, $H' \vdash F'$ **ok**, *and* $\forall B.\ H \leq_B H'$.
2. *If* $H \vdash FS : \sigma$, $H \vdash FS$ **ok** *and* $H, FS \twoheadrightarrow H', FS'$ *then* $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash FS' : \sigma$, $H' \vdash FS'$ **ok** *and* $H \leq_{obsIds(FS)} H'$.
3. *If* $H \vdash P : \star$, $H \vdash P$ **ok** *and* $H, P \rightsquigarrow H', P'$ *then* $\vdash H' : \star$, $\vdash H'$ **ok**, $H' \vdash P' : \star$ *and* $H' \vdash P'$ **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* $H \vdash F : \sigma$, $H \vdash F$ ***ok*** *and* $H, F \longrightarrow H', F'$ *then* $\vdash H' : \star$, $\vdash H'$ ***ok***, $H' \vdash F' : \sigma$, $H' \vdash F'$ ***ok***, *and* $\forall B.\ H \leq_B H'$.
2. *If* $H \vdash FS : \sigma$, $H \vdash FS$ ***ok*** *and* $H, FS \twoheadrightarrow H', FS'$ *then* $\vdash H' : \star$, $\vdash H'$ ***ok***, $H' \vdash FS' : \sigma$, $H' \vdash FS'$ ***ok*** *and* $H \leq_{obsIds(FS)} H'$.
3. *If* $H \vdash P : \star$, $H \vdash P$ ***ok*** *and* $H, P \rightsquigarrow H', P'$ *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, Millikin, Bracha. *Spicing Up Dart with Side Effects*. ACM Queue 13.3, 2015
- Meijer. *Your mouse is a database*. CACM 55.5, 2012
- Syme, Petricek, Lomov. *The F# Asynchronous Programming Model*. PADL 2011

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/

Thank you!

Results

Questions?

- 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/