Towards Type-safe Composition of Actors

Dominik Charousset, January 2016

Problem Statement

1. Actors lack (message) interfaces
2. Actors hard-code receivers
3. Actors do not compose

Actors lack Interfaces

- Erlang and Akka use dynamic typing for actors
  - No type checking of messages at sender
  - Burdens developer with correctness checking
- SALSA, Pony, Charm++, etc. use an OO design
  - Member function names identify message types
  - Static type checking but inflexible, tight coupling

Actors Hard-code Receivers

- Erlang-like implementations use explicit sends only
  - Request/response modeled via two sends
  - Programmers implement pipelining manually
- OO-inspired designs have call semantics
  - Request/response modeled via return values
  - Caller always receives results

Actors do not Compose

- No system offers configurable message flows
- Users cannot define actors in terms of other actors

Composability*

- A type is composable if instances of it can be combined to produce the same or a similar type
- Abstraction that enables gluing solutions together
    - Reuse existing components without modification
    - Modularize architecture

** Loosely based on the definition in "Why Functional Programming Matters"*

Composability in FP

- Functions are first-class citizens
  - Can be stored in variables / bound to names
  - Can be passed to (higher order) functions
- Functions use parametric polymorphism*
  - Generic while maintaining full static type-safety
  - Transformation of data types and other functions

** Allow deducing theorems for polymorphic functions, see "Theorems for free!"*

Dot Operator in Haskell

- `f.g` composition pipes the output of `g` to `f`
  - Output type of `g` is the input type of `f`
  - `f.g` has input type of `g` and output type of `f`
- Definition in the Haskell standard library:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

Dot Operator for Actors?

- Composition requires reasoning about types:
  - What input types have actors `F` and `G`? *
  - What output types generate `F` and `G`?
- Reasoning about types requires:
  - Unambiguously define input and output types
  - Force input -> output messaging style upon actors

** We denote actors using uppercase and functions using lowercase letters*

State of the Art

- Traditionally: dynamic typing, messages are tuples
  - Erlang, Scala Actors*, Akka
- Alternative approach: OO-inspired design
  - SALSA, Pony, Charm++

** Deprecated since Scala 2.11 in favor of Akka*

Traditional Design

- No correlation of input and output messages
- Hard-coded message receivers
- Atoms or case classes identify operations
- Akka also offers non-messaging future-based API

TAkka*

- Attempts to add type-safety to Akka
- Restricts input types for actors, but not outputs
- Embeds manifests for run-time type checks
- Keeps hard-coding of receivers as found in Akka

** See "Typecasting actors: from Akka to TAkka"*

OO-inspired Design

- Hides messages using named methods
- Defines both input and output types of methods
- Tightly couples caller and callee via type system
  - Caller needs to know type or supertype or callee
  - Structural types can prevent overly tight coupling

Structural Types

- Enable type-safe duck typing
- Hide actual type of callee but still bind to names
- Allow compiler to check compliance of interfaces
- Require *user-defined* definition of method
- Unsuitable for "dot operator"-style composition

Conceptual Idea

1. Correlate inputs and outputs without OO design
2. Enable the runtime to manipulate message flows
3. Offer minimal set of composition primitives
4. Compose actors from other actors

Expose Message Flow

- Actors define input and output tuples as interface
- Interfaces are sets of `(a, b) -> (c, d)` rules
- Uniquely typed atoms allow to identify operations
  - Example interface for an associative container:
    ```
    ('get', string) -> (int)
    ('set', string, int) -> ()
    ```

Manipulate Message Flow

- Prohibit actors from sending results manually
- Generate responses from message handler results
- Store messaging path in a message header
- Build pipelines by redirecting responses

Find Composition Primitives

- Allow users to alter actor interfaces (unary)
  - Pre-define or re-order input and output types
  - Create adapter interface for further composition
- Enable user to compose two actors (binary)
  - Compose result further with a third actor, etc.
  - Structure compositions like a tree

Compose Unary and Binary

- Unary composition: partial applications / bindings
  - Binding inputs: `H(x) = F(δ $ x)` *
  - Binding outputs: `H(x) = δ $ F(x)`
- Binary composition: sequential or parallel
  - Pipelines: `H(x) = F(G(x))`
  - Joins: `H(x) = (F(x), G(x))`

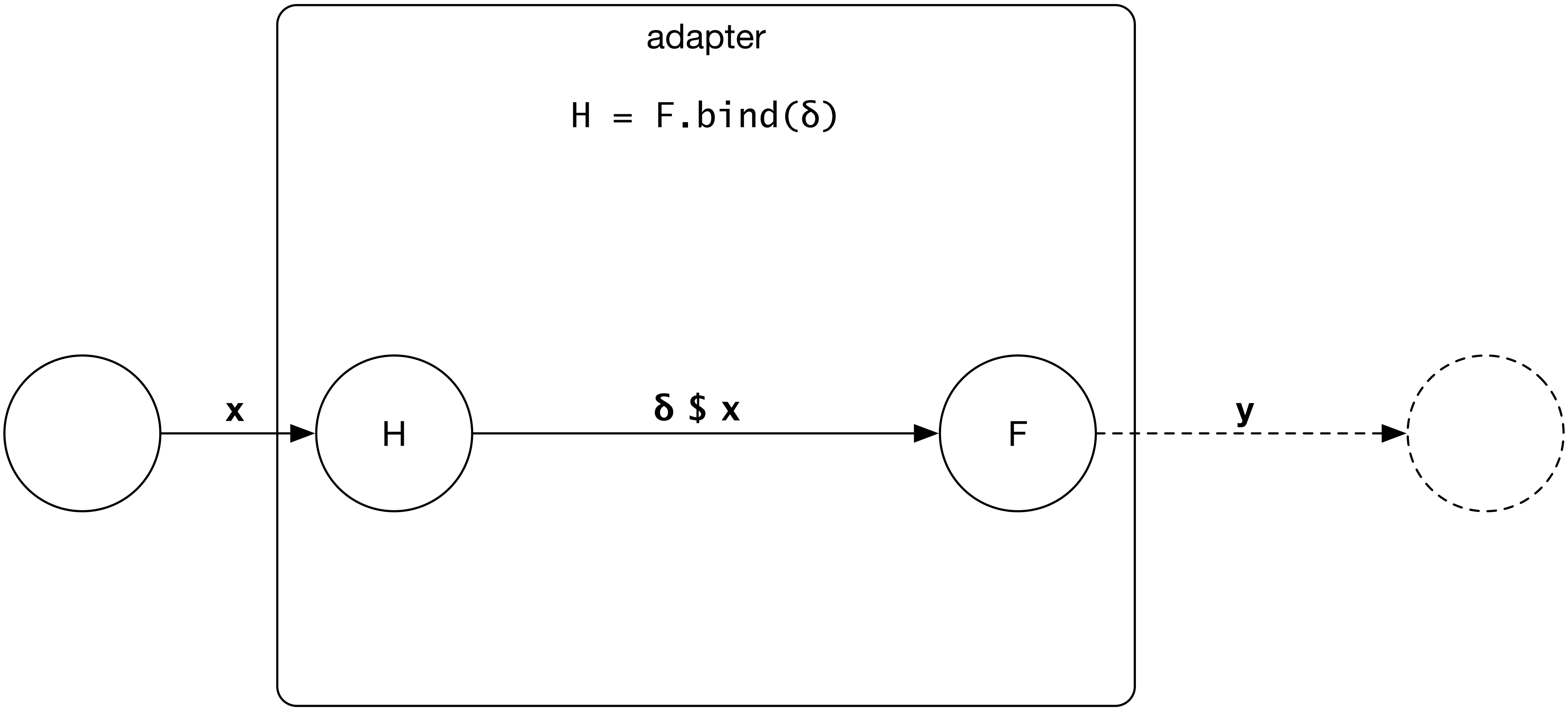
** `$` is the apply operator, `δ` is a user-defined bind operation*

Define Composed Actors

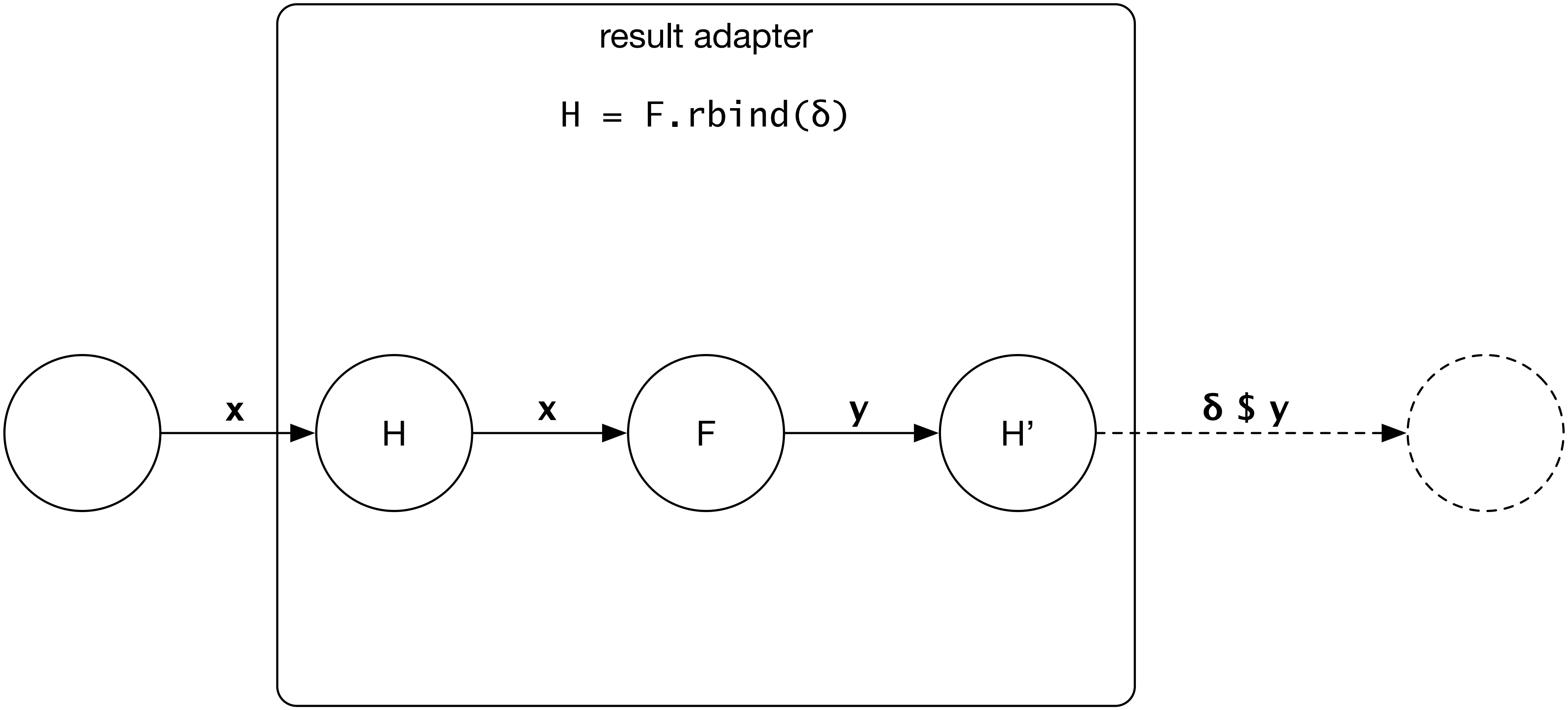
- Have their own identity
- Are never scheduled and have no mailbox
- Manipulate messages and message paths
- Can spawn actors for stateful operations (e.g. join)
- Cease to exist if any of their constituents dies

Implementation in CAF

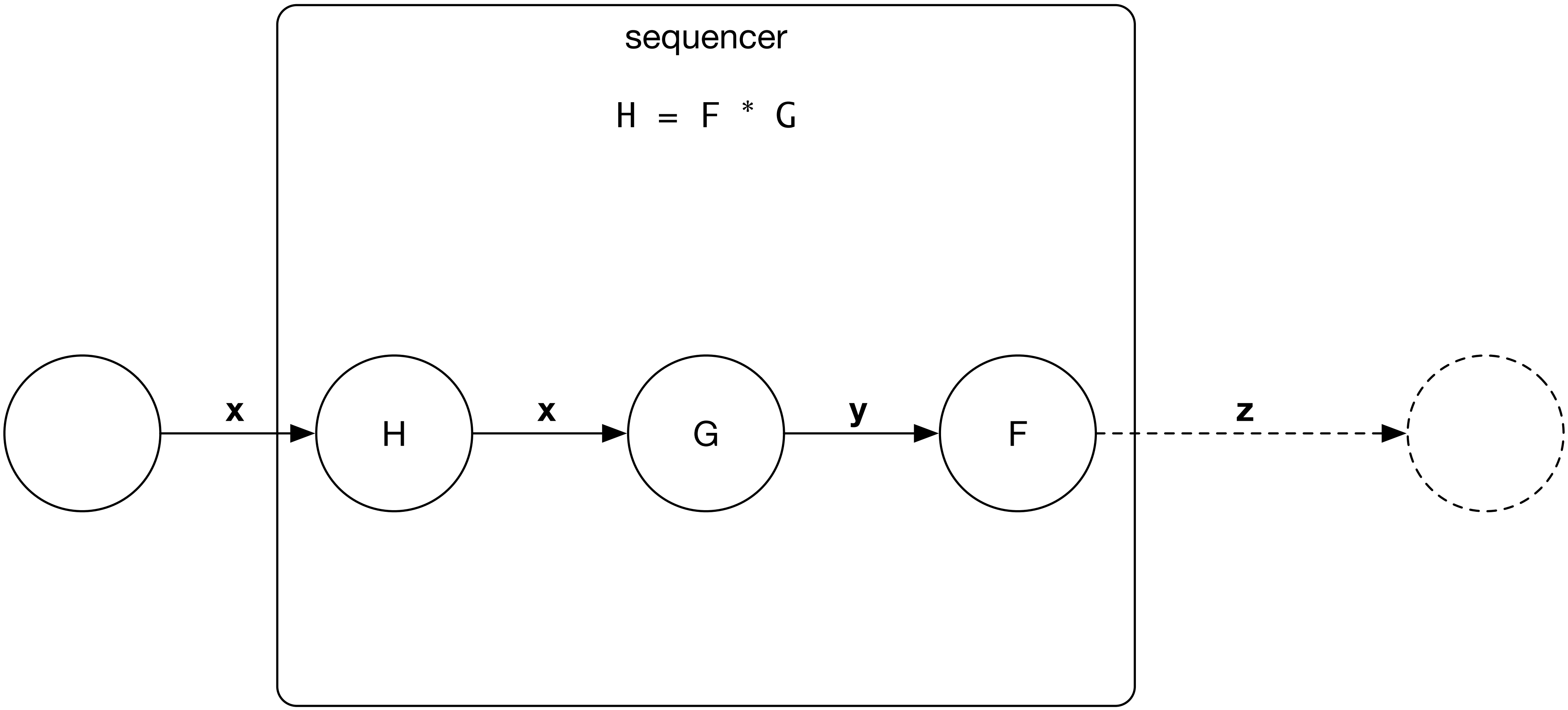
- Messaging interfaces based on tuples only
  - Abstract: `(a, b) -> (c, d)`
  - C++: `replies_to<a, b>::with<c, d>`
- Composition is implemented with 4 decorators
  - 2 for unary compositions
  - 2 for binary compositions



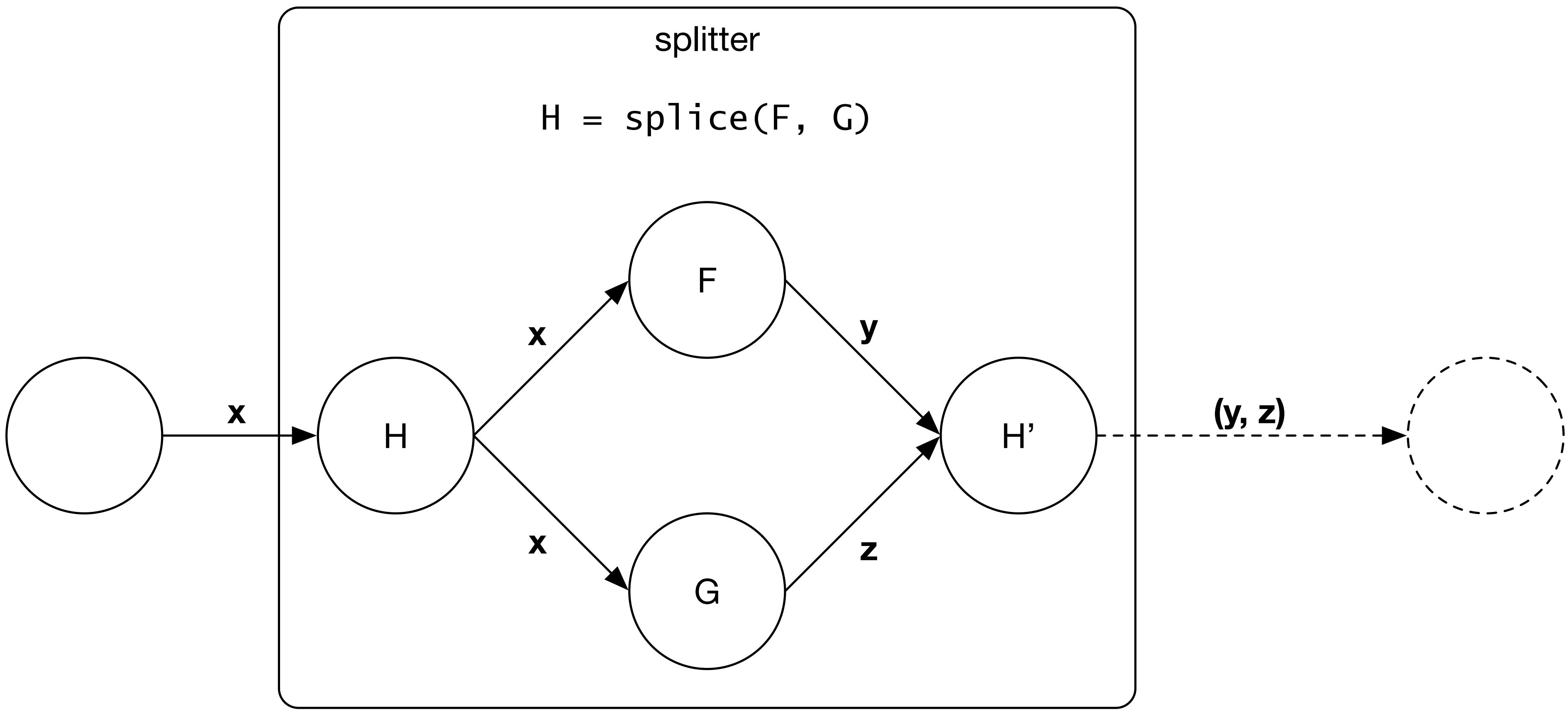
FP equivalent: $h = \lambda(x) \rightarrow f(\delta \ \$ \ x)$



FP equivalent: $h = \lambda(x) \to \delta \ \$ \ f(x)$



FP equivalent: $h = f \cdot g$



FP equivalent: `h = λ(x) -> (f(x), g(x))`

Types of Composed Actors

- Unary composition:
  - Remove types for partial applications
  - Select all possible clauses for reordering via bind
- Binary composition:
  - Sequencer accept `in(G)`, return `F(G(x))`
  - Splitter accept `in(F) ∩ in(G)`, return `(F(x), G(x))`

* *`in(F)` denotes all accepted input tuples for `F`*

Currying

calculator

```
(‘add’, int, int) -> (int)
(‘sub’, int, int) -> (int)
(‘mul’, int, int) -> (int)
(‘div’, int, int) -> (int)
```

```
.bind(‘add’, _1, _2)
```

=

adder

```
(int, int) -> (int)
```

Binding

calculator

```
(‘add’, int, int) -> (int)
(‘sub’, int, int) -> (int)
(‘mul’, int, int) -> (int)
(‘div’, int, int) -> (int)
```

```
.bind(‘mul’, _1, 2)
```

=

doubler

```
(int) -> (int)
```

Prefixing Results

calculator

```
(int) -> (int)
(int, int) -> (int, int)
```

```
.rbind(‘res’, _1)
```

=

adder

```
(int) -> (‘res’, int)
(int, int) -> (‘res’, int)
```

Pipelining

F		G		H

`(int) -> (double)`
`(string, string) -> (string)`
	∘	`(int, string) -> (int)`
		`(int, double) -> (string, int)`
		`(string) -> (string, string)`
			=	`(int, string) -> (double)`
				`(string) -> (string)`

Joining

F

```
(int) -> (double)
(string) -> (string)
```

++

G

```
(int) -> (int, int)
(string) -> (int)
(string, string) -> (bool)
```

=

H

```
(int) -> (double, int, int)
(string) -> (string, int)
```

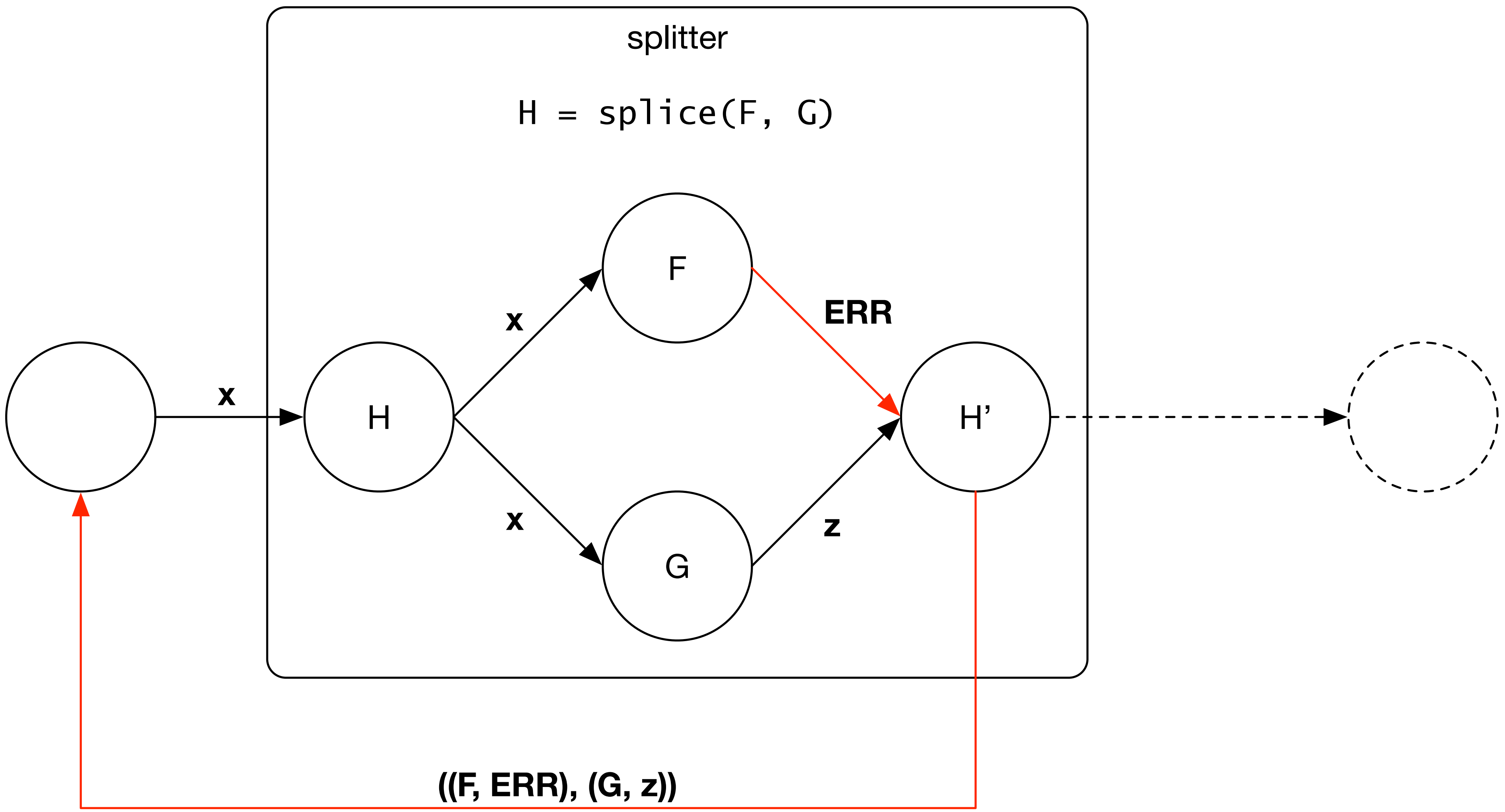
Error Handling

- Escalate errors always to original sender
- Indicate at what stage error occurred
  - Trivial for sequencers: either `F` or `G` sends error
  - Splitters have 3 possible error states

Error States of Splitters

1. `F` failed, but not `G`
2. `G` failed, but not `F`
3. Both `F` and `G` failed

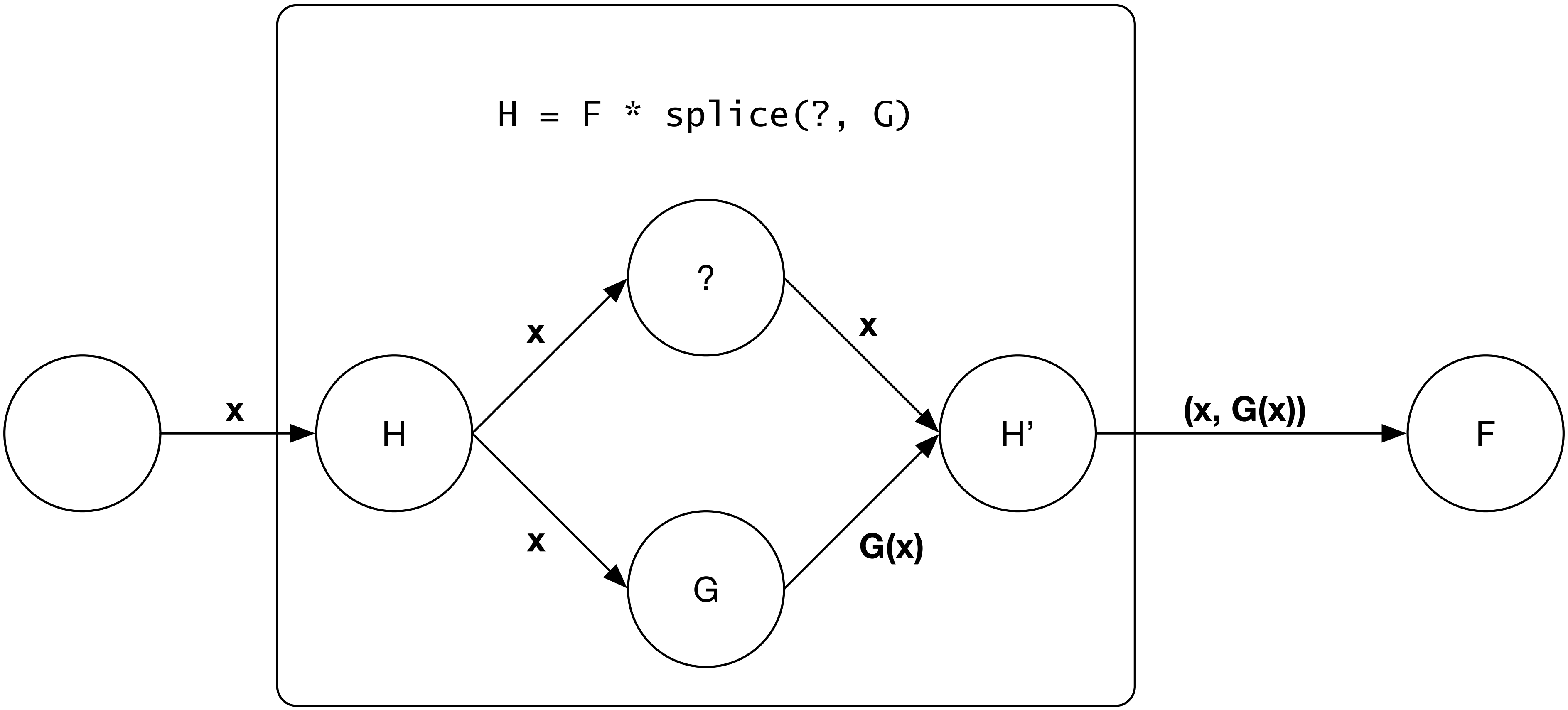
➡ Transmit result of `F` and `G` regardless of error



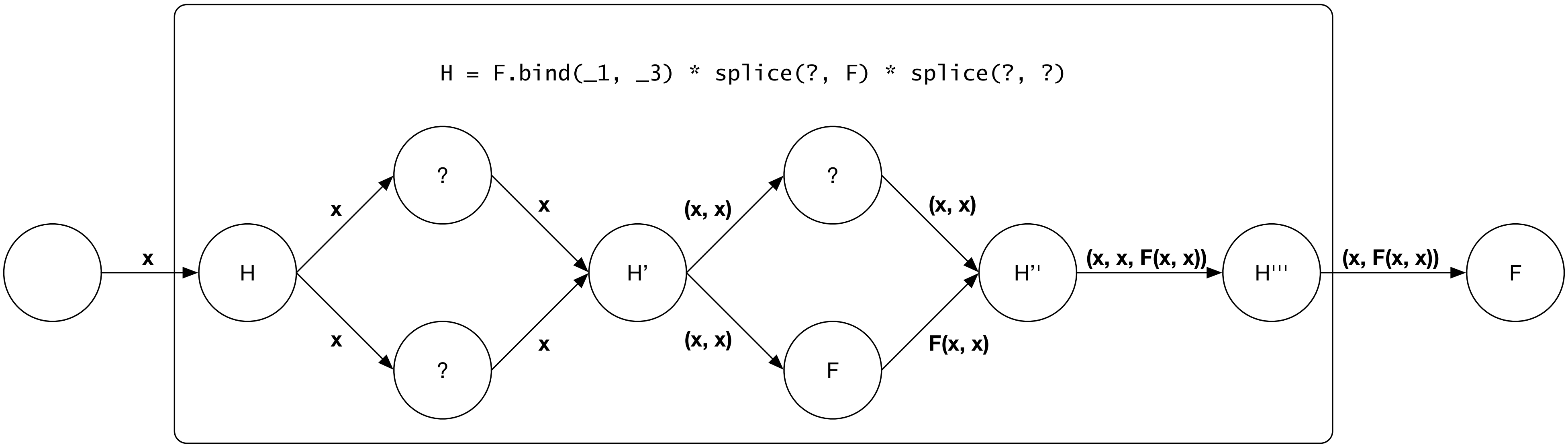
splitter
H = splice(F, G)
F
ERR
x
x
H
H’
x
z
G
((F, ERR), (G, z))

Re-using Inputs

- Consider
  - `H(x) = F(x, G(x))`
  - `H(x) = F(x, F(x, x))`
- How to multiply inputs for later stages?



FP equivalent: $h = \lambda(x) \rightarrow f(x, g(x))$



FP equivalent: h = λ(x) -> f(x, f(x, x))

Idea: Pseudo Actors

- Enable more complex composition
- From the examples: `?` denotes the "identity actor"
- Have polymorphic interface (depends on the input)

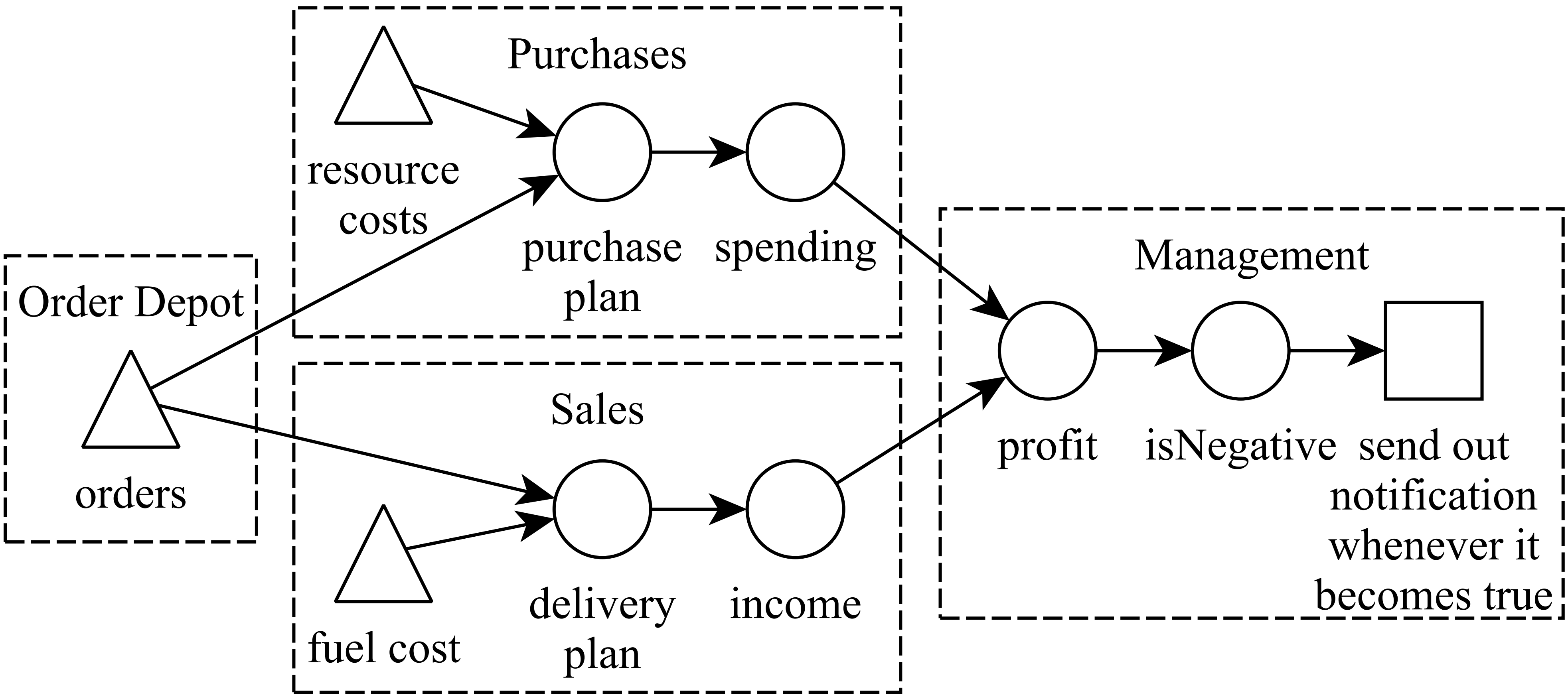
Efficiency

- Not the goal of this work, but a side effect
- Concurrent setting: fewer scheduler cycles
- Distributed setting: fewer inter-node messages
  - Composed actors are considered as `values`
  - Never referenced across nodes, always copied

Reactive Programming RP

- Type-safe composable actors overlap with RP
- Accumulated updates are glitch-free by design

Example RP Case Study



** from "Distributed REScala: An Update Algorithm for Distributed Reactive Programming"*

Conclusion

- We have a more functional take on actors
- Composability increases expressive power of CAF
- New API has overlap to distr. reactive programming
- More robust and efficient than user-generated code

Future Work

- Pseudo actors with polymorphic interfaces
- Higher-level buildings blocks based on primitives
- Explore connections and tradeoffs to RP
  - Compare generated code / "ease of use"
  - Evaluate differences in performance/networking

Thank you for your attention!

Questions?

References

- Hughes, J., "Why Functional Programming Matters", Vol. 32 of Computer Journal, pp. 98-107, Oxford University Press, Oxford, UK, 1989
- Wadler, Philip. "Theorems for free!", in Proceedings of the fourth international conference on Functional programming languages and computer architecture, pp. 347-359. ACM, 1989
- He, Jiansen, Philip Wadler, and Philip Trinder. "Typecasting actors: from Akka to TAkka." In Proceedings of the Fifth Annual Scala Workshop, pp. 23-33. ACM, 2014.
- Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. "Distributed REScala: an update algorithm for distributed reactive programming." In Proceedings of the 2014 ACM OOPSLA '14, pp. 361-376 ACM, 2014.