Concurrency Theory
vs
Concurrent Languages

Silvia Crafa

Universita' di Padova

Bertinoro, OPCT 2014

Bisimulation inside

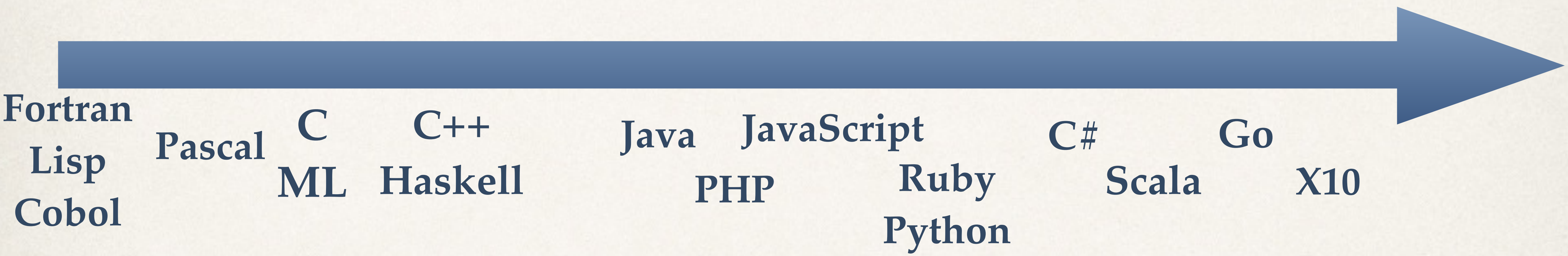
Concurrency Theory vs Concurrent Languages

Silvia Crafa

Universita' di Padova

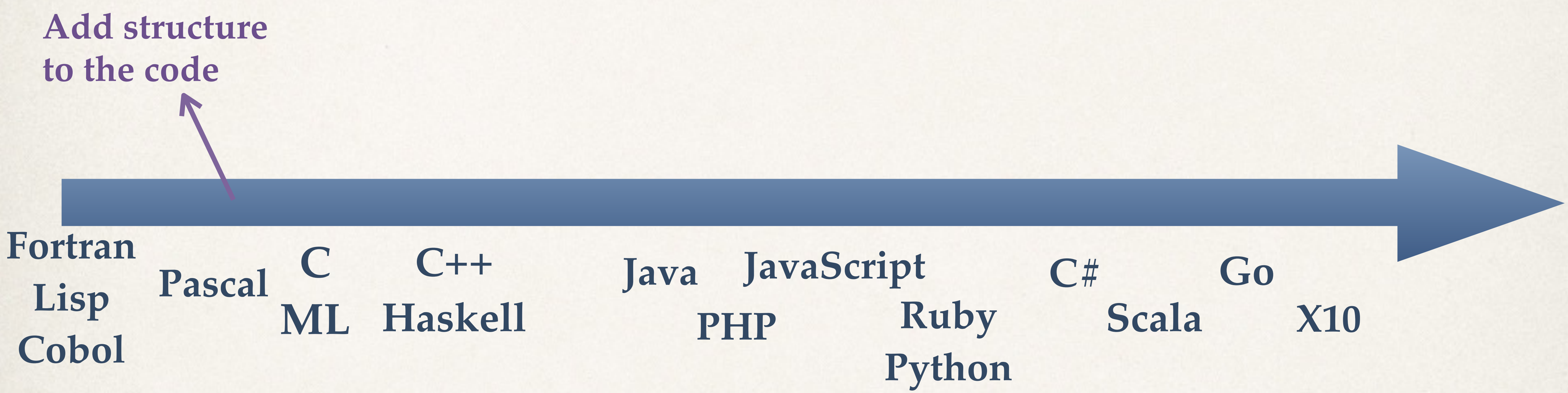
Bertinoro, OPCT 2014

The Quest for good Abstractions



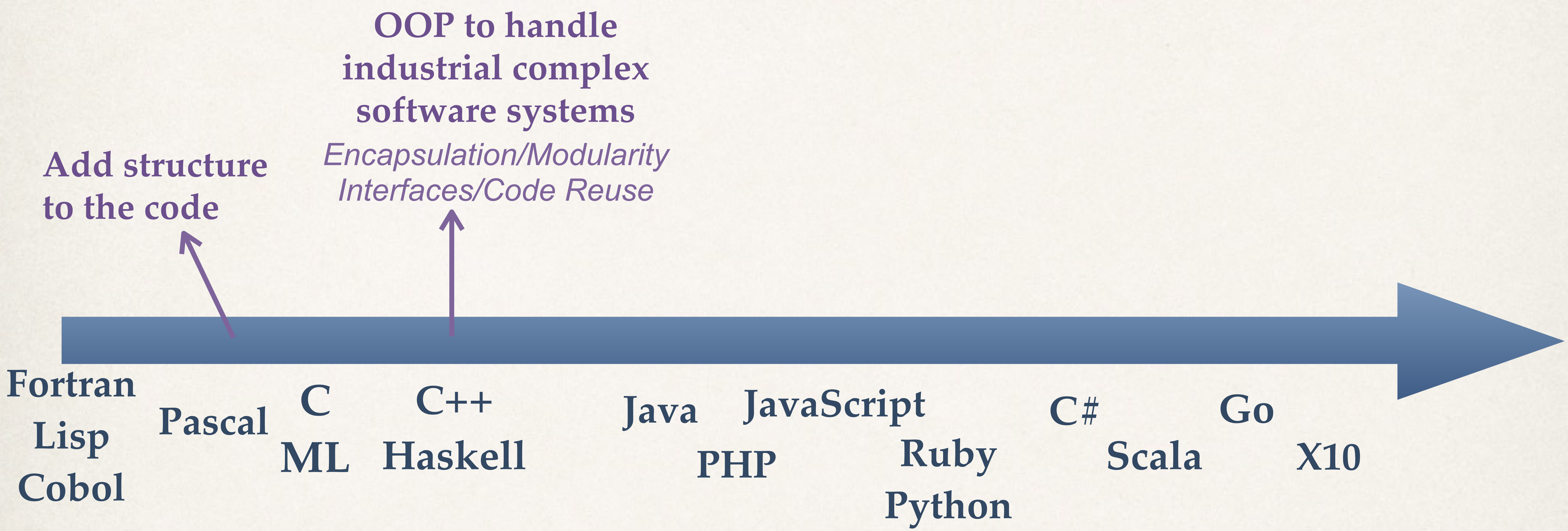
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



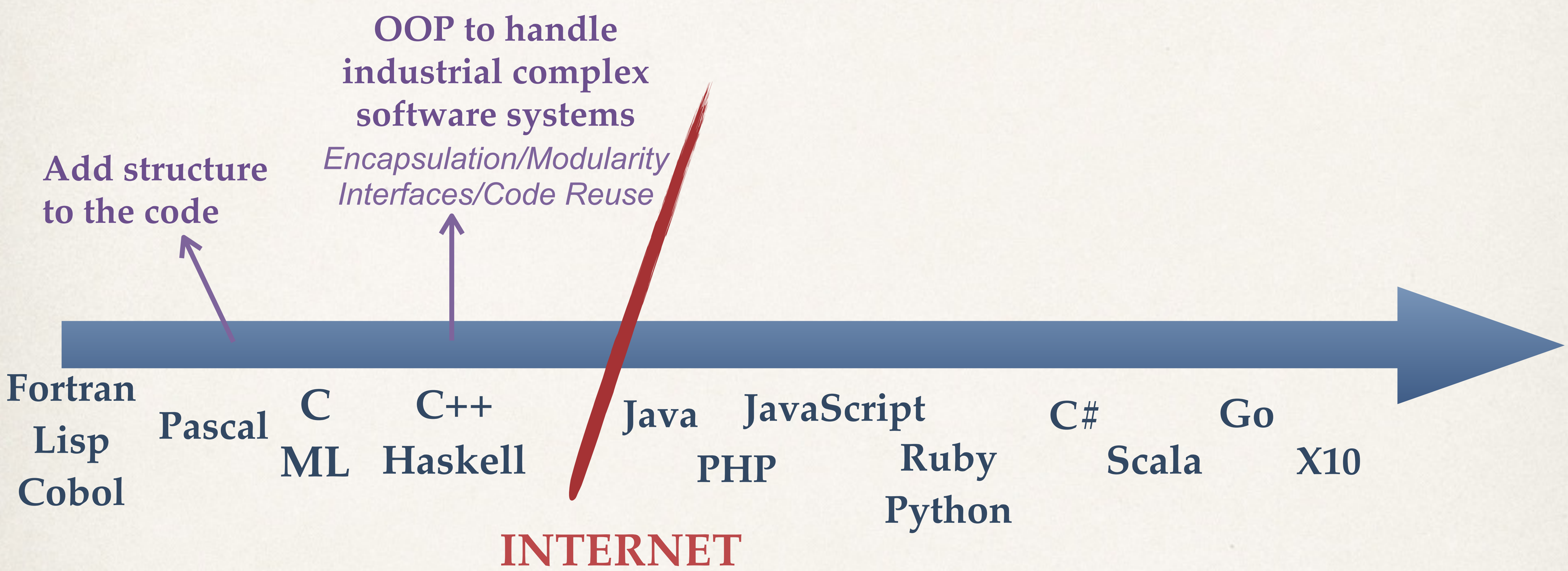
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



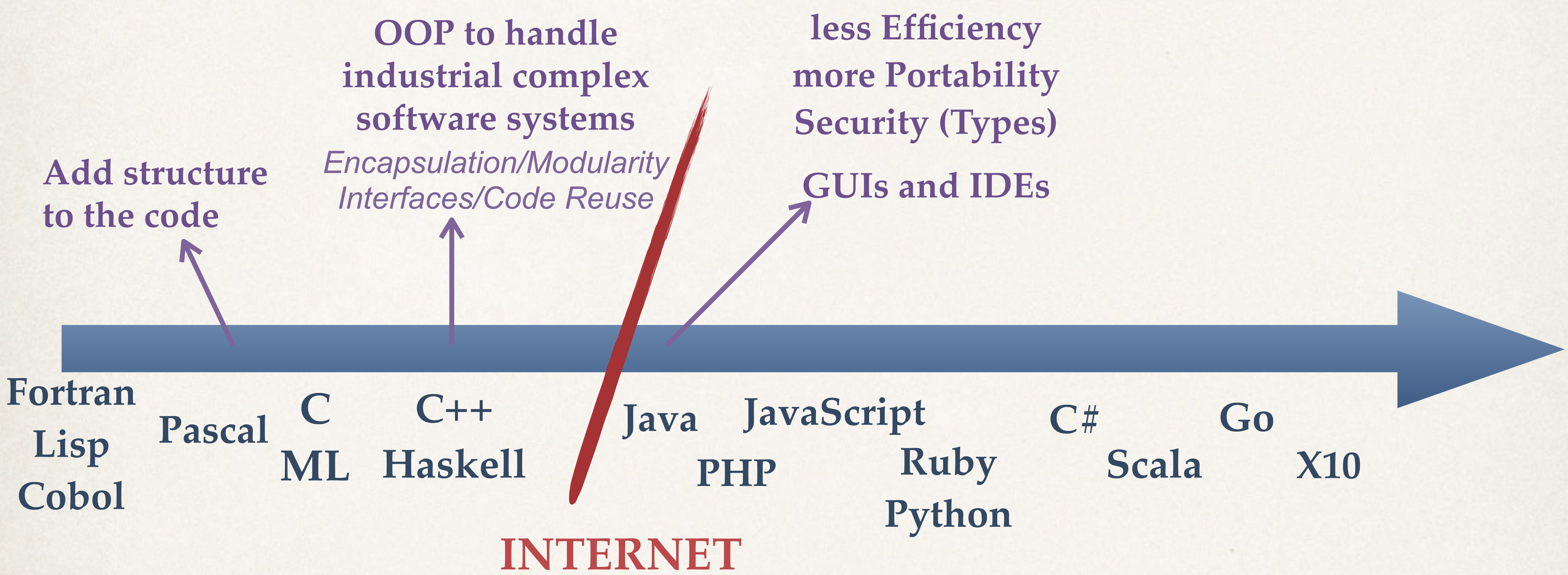
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



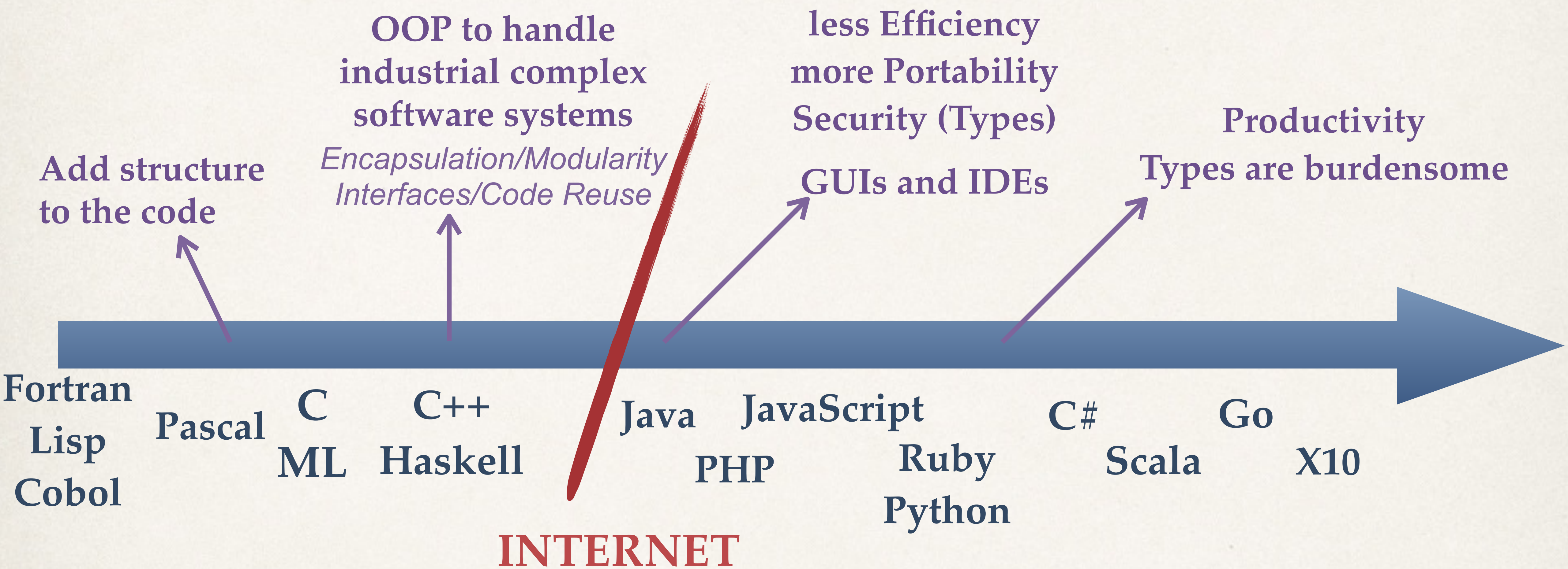
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



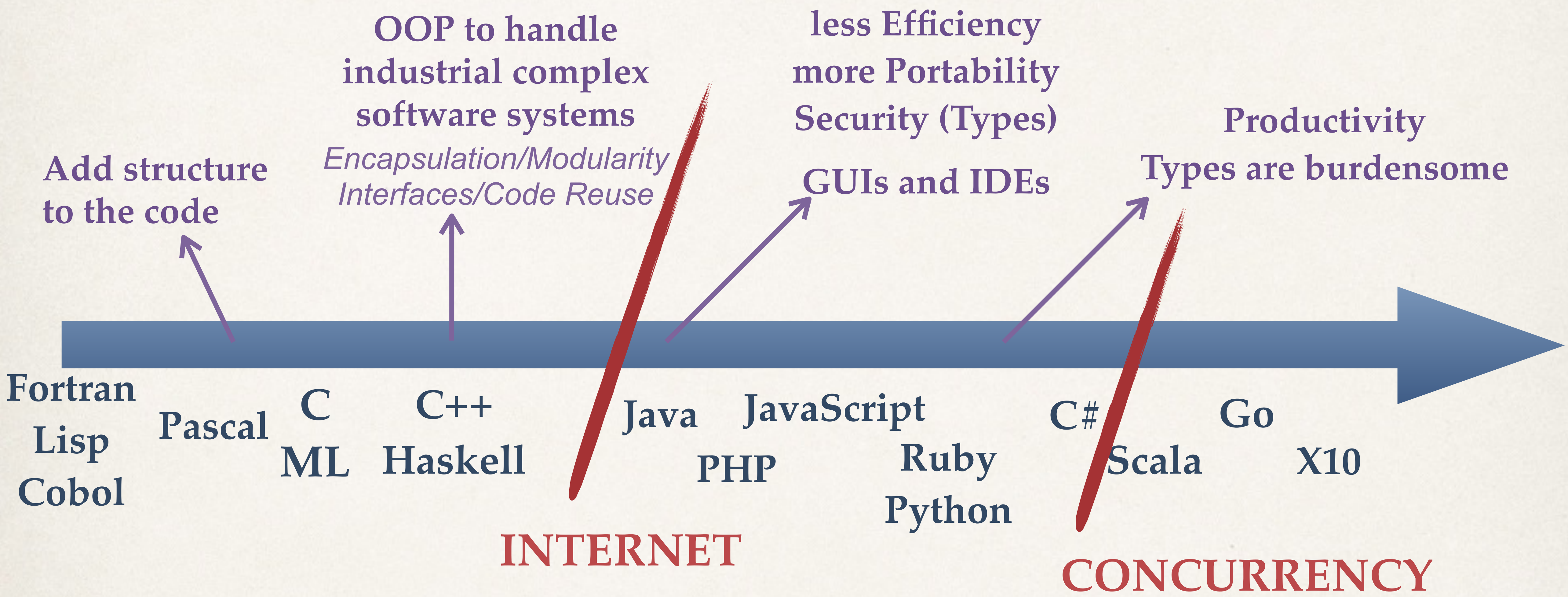
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



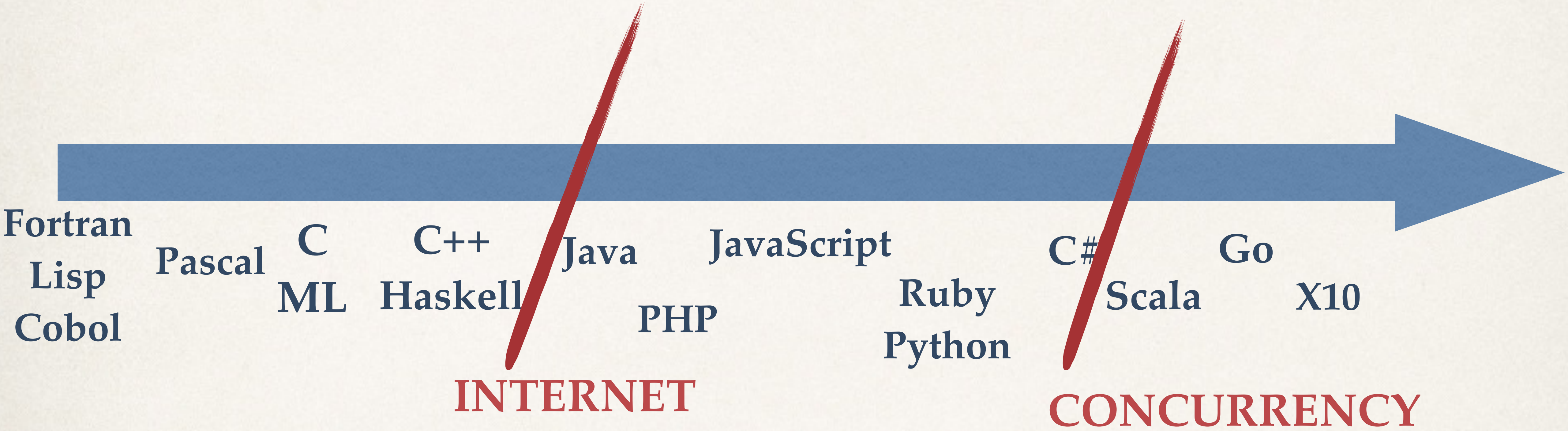
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



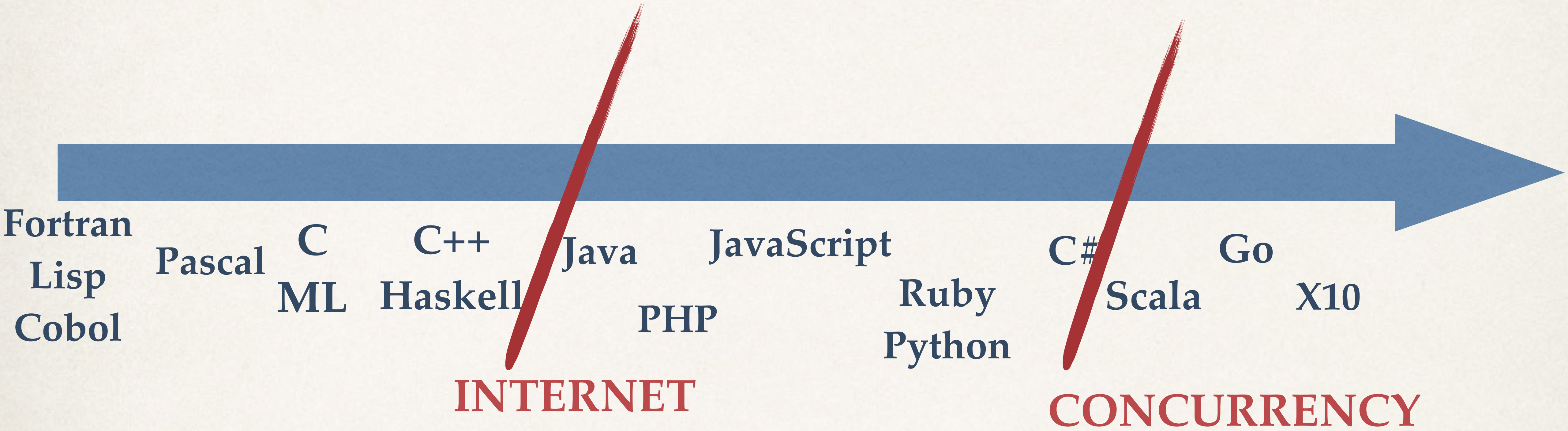
- **When** a language has been **invented** VS when became **popular**?
- **Why** has been invented VS why became popular?

The Quest for good Abstractions



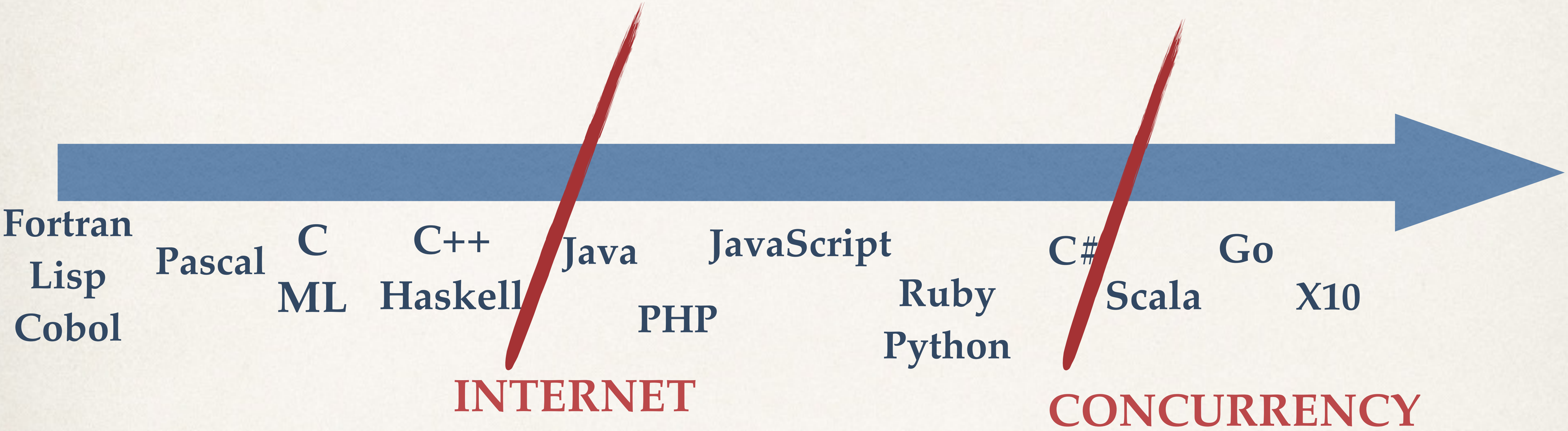
- **<u>Changes need a catalyser</u>**

The Quest for good Abstractions



- **Changes need a catalyser**
  - **new hardware can only be parallel**
  - new software must be concurrent

The Quest for good Abstractions



- **Changes need a catalyser**
  - **new hardware can only be parallel**
  - new software must be concurrent

Popular Parallel Programming Grand Challenge

How hard is Concurrent Programming?

- (correct) concurrent programming is **difficult**
- **Adding** concurrency to sequential code is even harder

Intrinsic reasons
nondeterminism

Accidental reasons
improper
programming model

How hard is Concurrent Programming?

- (correct) concurrent programming is **difficult**
- **Adding** concurrency to sequential code is even harder

Intrinsic reasons
nondeterminism

Accidental reasons
improper
programming model

***Think* concurrently**
(Concurrent Algorithm)

↔

***Translate* into**
a concurrent code

How hard is Concurrent Programming?

- (correct) concurrent programming is **difficult**
- **Adding** concurrency to sequential code is even harder

Intrinsic reasons
nondeterminism

Accidental reasons
improper
programming model

***Think* concurrently**
(Concurrent Algorithm)

***Translate* into**
a concurrent code

DESIGN of
concurrent language

How hard is Concurrent Programming?

- (correct) concurrent programming is **difficult**
- **Adding** concurrency to sequential code is even harder

Intrinsic reasons
nondeterminism

Accidental reasons
improper programming model

***Think* concurrently**
(Concurrent Algorithm)

***Translate* into**
a concurrent code

DESIGN of concurrent language

High-level Concurrency Abstraction

The Quest for good Abstractions

Easy to think
Easy to reason about

↔

Expressiveness
Performance

The Quest for good Abstractions

Easy to think
Easy to reason about ⟷ **Expressiveness**
Performance

- **OOP**
  - encapsulation
  - memory management
  - multiple inheritance

The Quest for good Abstractions

Easy to think
Easy to reason about ⟷ **Expressiveness**
Performance

- **OOP**
  - encapsulation
  - memory management
  - multiple inheritance

C++ —> Java —> Scala

The Quest for good Abstractions

Easy to think
Easy to reason about

⟷

Expressiveness
Performance

- **OOP**
  - encapsulation
  - memory management
  - multiple inheritance

C++ —> Java —> Scala

- **Types**
  - documentation vs verbosity

C++ —> Java —> Ruby —>Scala

The Quest for good Abstractions

Easy to think
Easy to reason about ⟷ **Expressiveness**
Performance

- **OOP**
  - encapsulation
  - memory management
  - multiple inheritance

C++ —> Java —> Scala

- **Types**
  - documentation vs verbosity

C++ —> Java —> Ruby —>Scala

- **Functional Programming**
  - composing and passing behaviours
  - sometimes imperative style is easier to reason about

C#—> Scala C++11, Java8

The Quest for good Abstractions

Easy to think
Easy to reason about

↔

Expressiveness
Performance

- OOP

which abstractions <u>interoperate</u> productively?

- Types

- Functional Programming

The Quest for good Abstractions

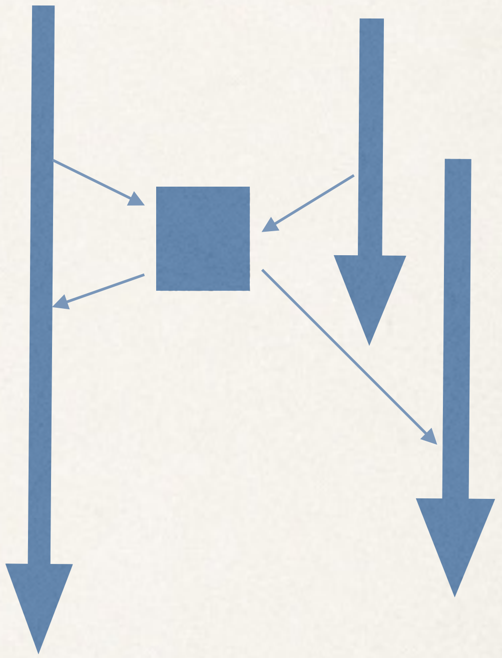
Concurrency Abstractions?

Many Concurreny Models…

The Quest for good Abstractions

Concurrency Abstractions?

Many Concurreny Models…



- Shared Memory Model and “Java Threads”

The Quest for good Abstractions

Concurrency Abstractions?

Many Concurreny Models…

- Shared Memory Model and "Java Threads"

Java

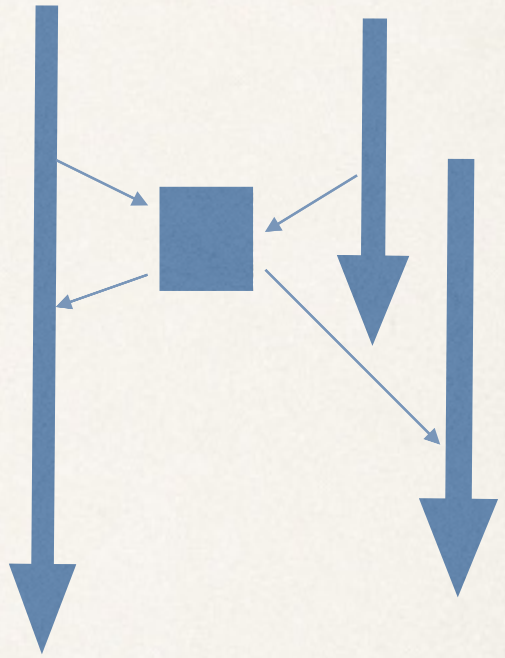
```
synchronized(lock)
lock.wait()
lock.notify()
```

STM

```
atomic {…}
when(cond){…}
```

X10

```
async{}
finish{}
```

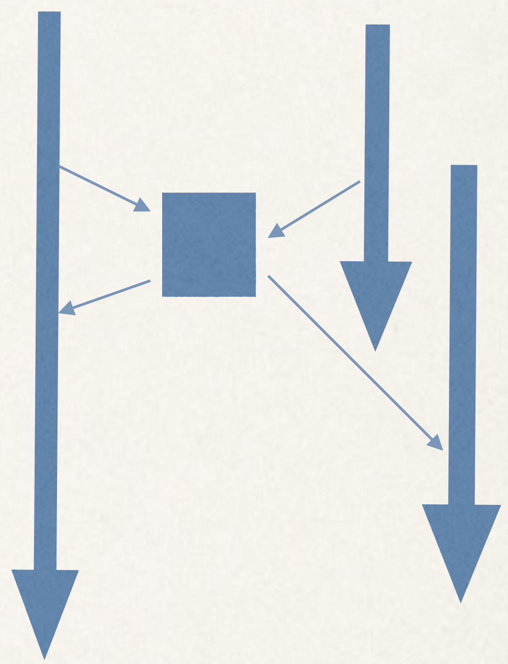


The Quest for good Abstractions

Concurrency Abstractions?

Many Concurreny Models…

- Shared Memory Model and “Java Threads”



Java

```
synchronized(lock)
lock.wait()
lock.notify()
```

STM

```
atomic {…}
when(cond){…}
```

X10

```
async{}
finish{}
```

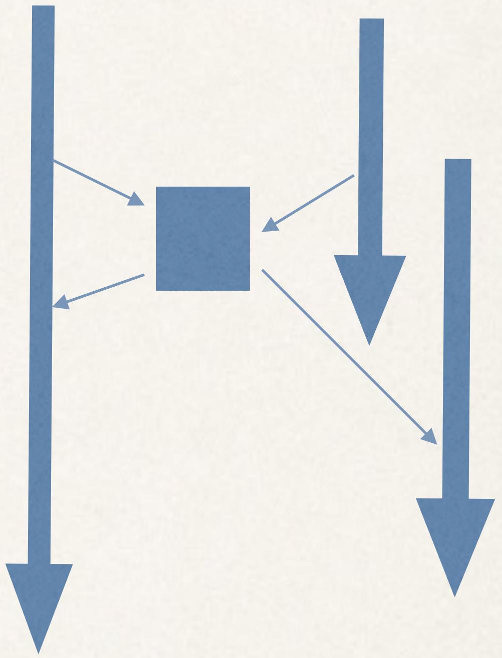
- **logical threads** distinguished from **executors**
  (activities/tasks) (pool of thread workers)

Scalability!

The Quest for good Abstractions

Concurrency Abstractions?

Many Concurreny Models…



- Shared Memory Model and "Java Threads"

Java

```
synchronized(lock)
lock.wait()
lock.notify()
```

STM

```
atomic {…}
when(cond){…}
```

X10

```
async{}
finish{}
```

```
new Thread().start()
```

JVM thread

Lightweight threads in the program
Pool of Executors in the runtime

Scalability!

- **logical threads** distinguished from **executors**
  (activities/tasks) (pool of thread workers)

Many Concurrency Models

- **Shared Memory**
  - is very natural for "centralised algorithms" and components operating on shared data
  - is error-prone when the sole purpose of SM is thread communication
- **Message Passing Model**
  - It is the message that carries the state!
  - **Channel based**: Google's GO
  - **Actor Model**: Erlang, Scala. It fits well both OOP and FP
  - **Sessions**
- **GPU Concurrency Model**
  - Massive data parallelism
  - integration with high-level concurrent language (X10, Nova, Scala heterogeneous compiler)

Many Concurrency Models

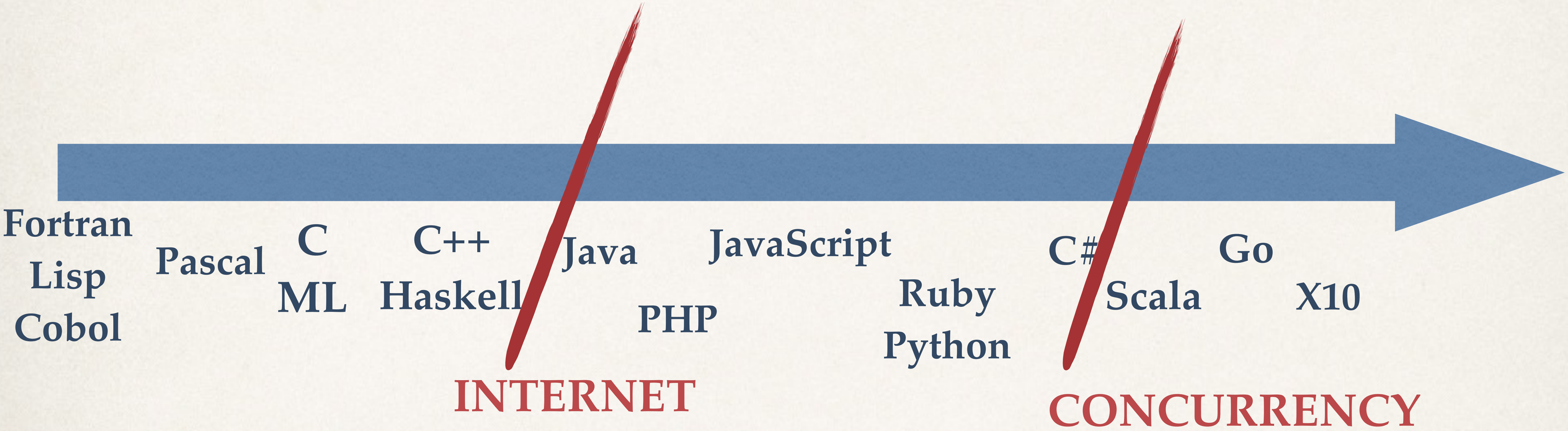
- Shared Memory

which abstractions <u>interoperate</u> productively?

- Message Passing Model

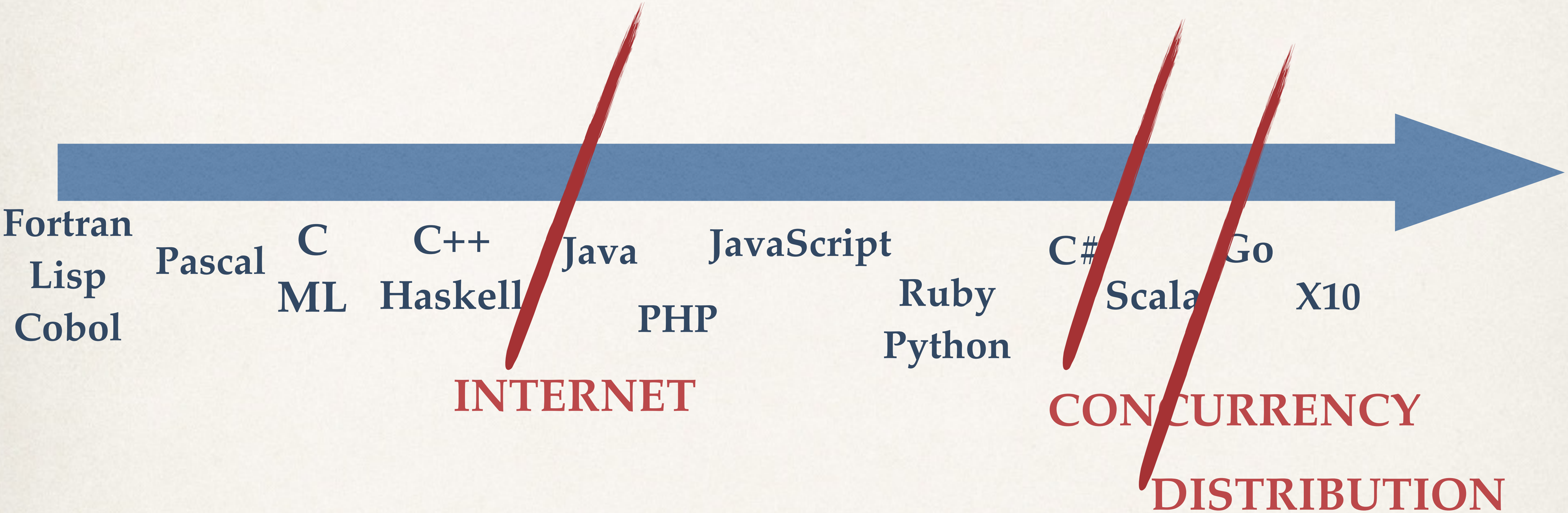
- GPU Concurrency Model

The Quest for good Abstractions



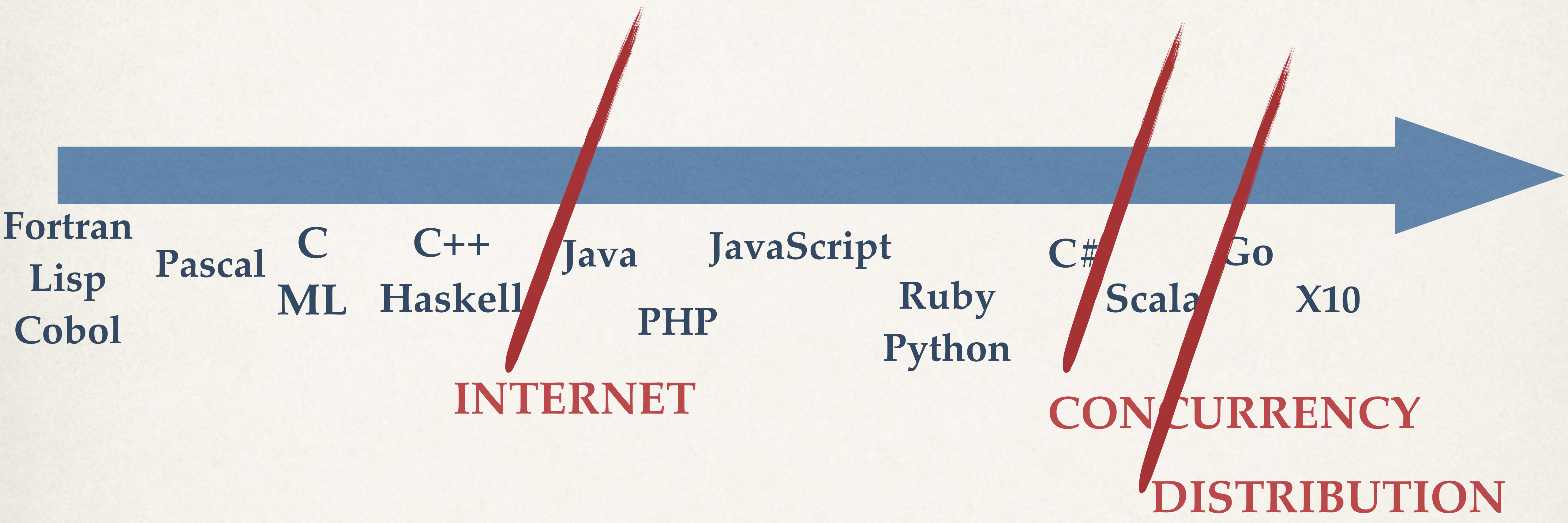
- New **catalyser**:

The Quest for good Abstractions



- New **catalyser**:
  - multicore —> concurrent programming
  - **cloud computing —> distributed programming**

The Quest for good Abstractions



- New **catalyser**:
  - multicore —> concurrent programming
  - **<u>cloud computing</u> —> distributed programming**

Reactive Programming

Reactive Programming

- react to events
- react lo load
- react to failures

Reactive Programming

- react to events
  - **event - driven**
  - **asynchronous**

instead of *issuing a command that asks for a change*, *react to an event that indicates that something has changed*

- **futures**
- **push data** to consumers when available rather than polling

- react lo load

- react to failures

Reactive Programming

- react to events
  - **event - driven**
  - **asynchronous**
- react lo load
  - **scalability**
    - up/down +/- CPU nodes
    - in/out +/- server
- react to failures

instead of *issuing a command that asks for a change*, *react to an event that indicates that something has changed*

- **futures**
- **push data** to consumers when available rather than polling

Reactive Programming

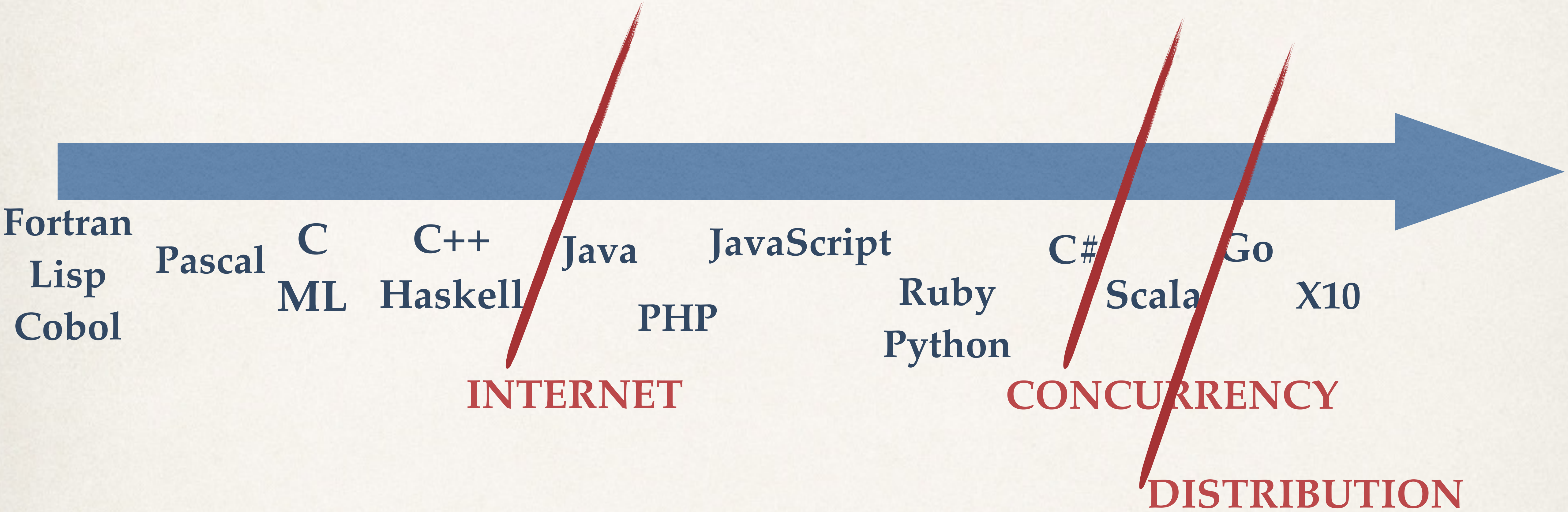
- react to events
  - **event - driven**
  - **asynchronous**

> instead of ***issuing a command that asks for a change***, ***react to an event that indicates that something has changed***

- react lo load
  - **scalability**
    - up/down +/- CPU nodes
    - in/out +/- server
- react to failures
  - **resiliency**

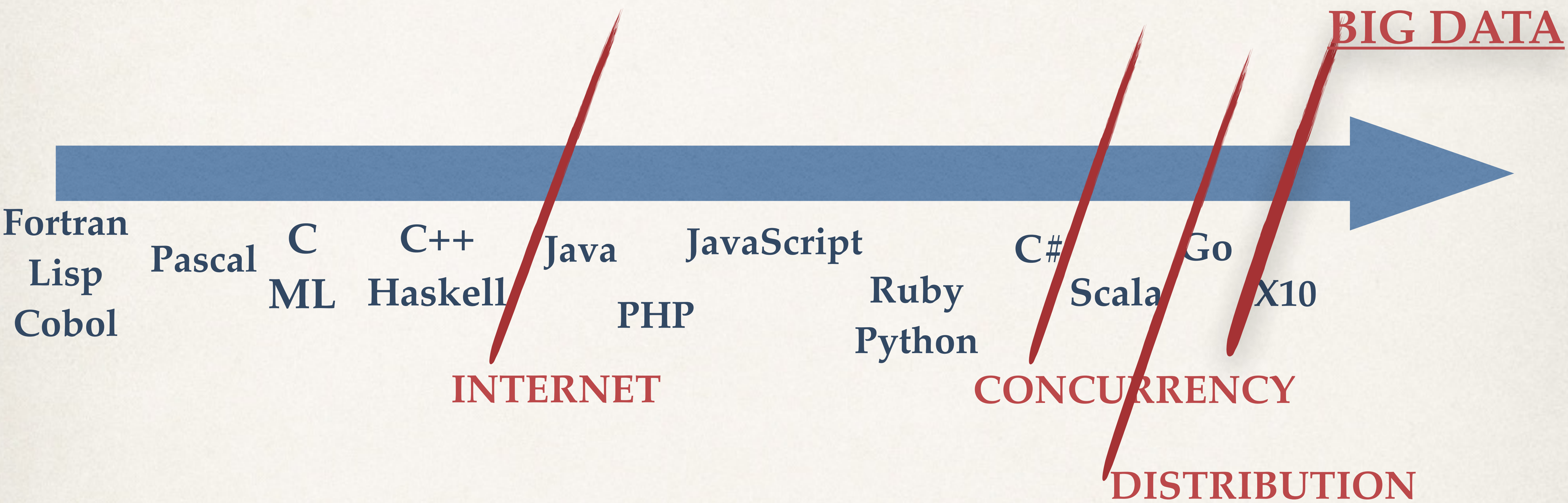
- **futures**
- **push data** to consumers when available rather than polling

The Quest for good Abstractions



- New **catalyser**:
  - multicore —> concurrent programming
  - cloud computing —> distributed programming

The Quest for good Abstractions



- New **catalyser**:
  - multicore —> concurrent programming
  - cloud computing —> distributed programming
  - **big data application —> High Performance Computing**

High Performance Computing

- scale-out on massively parallel hardware
  - high-performance computing on supercomputers
  - analytic computations on big data
- a single program
  - runs on a collection of places on a cluster of computers
  - can create global data-structures spanning multiple places
  - can spawn tasks at remote places, detecting termination of arbitrary trees of spawned tasks

High Performance Computing

- scale-out on massively parallel hardware
  - high-performance computing on supercomputers
  - analytic computations on big data
- a single program
  - runs on a collection of places on a cluster of computers
  - can create global data-structures spanning multiple places
  - can spawn tasks at remote places, detecting termination of arbitrary trees of spawned tasks

- **Big Data Application Framework**
  - **Map - Reduce Model**
  - **Bulk Synchronous Parallel Model**

High Performance Computing

- scale-out on massively parallel hardware
  - high-performance computing on supercomputers
  - analytic computations on big data
- a single program
  - runs on a collection of places on a cluster of computers
  - can create global data-structures spanning multiple places
  - can spawn tasks at remote places, detecting termination of arbitrary trees of spawned tasks

- **Big Data Application Framework**
  - **Map - Reduce Model**
  - **Bulk Synchronous Parallel Model**

"Concurrent Patterns"
with their
distinctive abstractions

What about Theory ?

The X10 experience

IBM Research

The X10 programming language

- open-source language for HPC programming

- key design features:
  - **scaling**: code running on 100 - 10.000 multicore nodes (up to 50millions core)
  - **productivity**: high level abstractions (Java-like, Scala-like) + typing (constrained dependent types as contracts).
  - **performance on heterogeneous hardware**: it compiles to Java, to C++, to CUDA. Resilient extension
  - **concurrent abstractions**: *place-centric, asynchronous computing*

The X10 programming language

```
// double in parallel all the array elements
val a:Array[Int]= …
      for(i in 0..(a.size-1))
            async { a(i)*=2 }
println ("The End")
```

Spawns an asynchronous lightweight activity running in parallel

The X10 programming language

```
 // double in parallel all the array elements
 val a:Array[Int]= …
finish for(i in 0..(a.size-1))
          async { a(i)*=2 }
 println ("The End")
```

waits for the termination
of all the spawned activities

Spawns an asynchronous
lightweight activity
running in parallel

The X10 programming language

```
// double in parallel all the array elements
val a:Array[Int]= …
var b=0
finish for(i in 0..(a.size-1))
    async { a(i)*=2
            atomic { b=b+a(i) }
          }
println (“The End”)
```

STM `when(cond) s`

clocks

The X10 programming language

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    finish for (p in Place.places())
       async at(p)
          Console.OUT.println("Hello from place "+p)
    Console.OUT.println("Hello from everywhere")
}
```

The X10 programming language

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    finish for (p in Place.places())
      async at(p)
        Console.OUT.println(“Hello from place “+p)
    Console.OUT.println(“Hello from everywhere”)
}
```

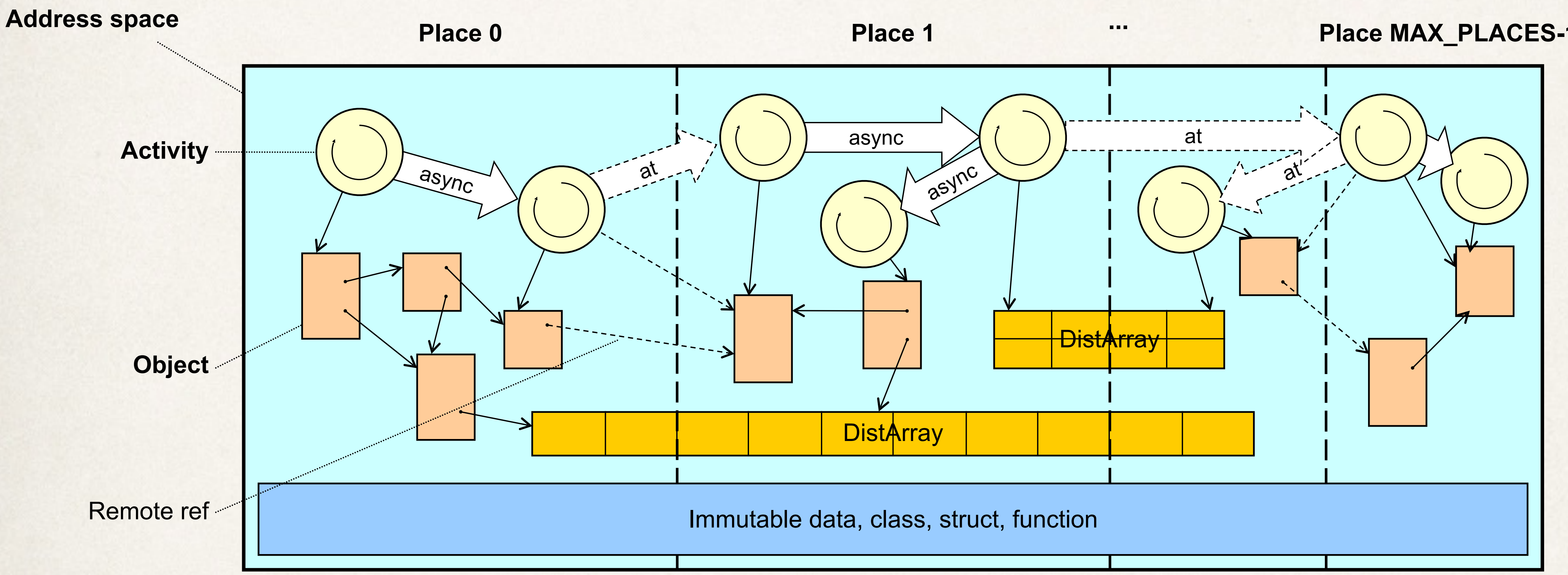
```
%X10_NPLACES=4
Hello from place 1
Hello from place 2
Hello from place 0
Hello from place 3
Hello from everywhere
```

The X10 programming language

```
class HelloWholeWorld {
  public static def main(args:Rail[String]) {
    finish for (p in Place.places())
      async at(p) @CUDA
        Console.OUT.println("Hello from place "+p)
    Console.OUT.println("Hello from everywhere")
}
```

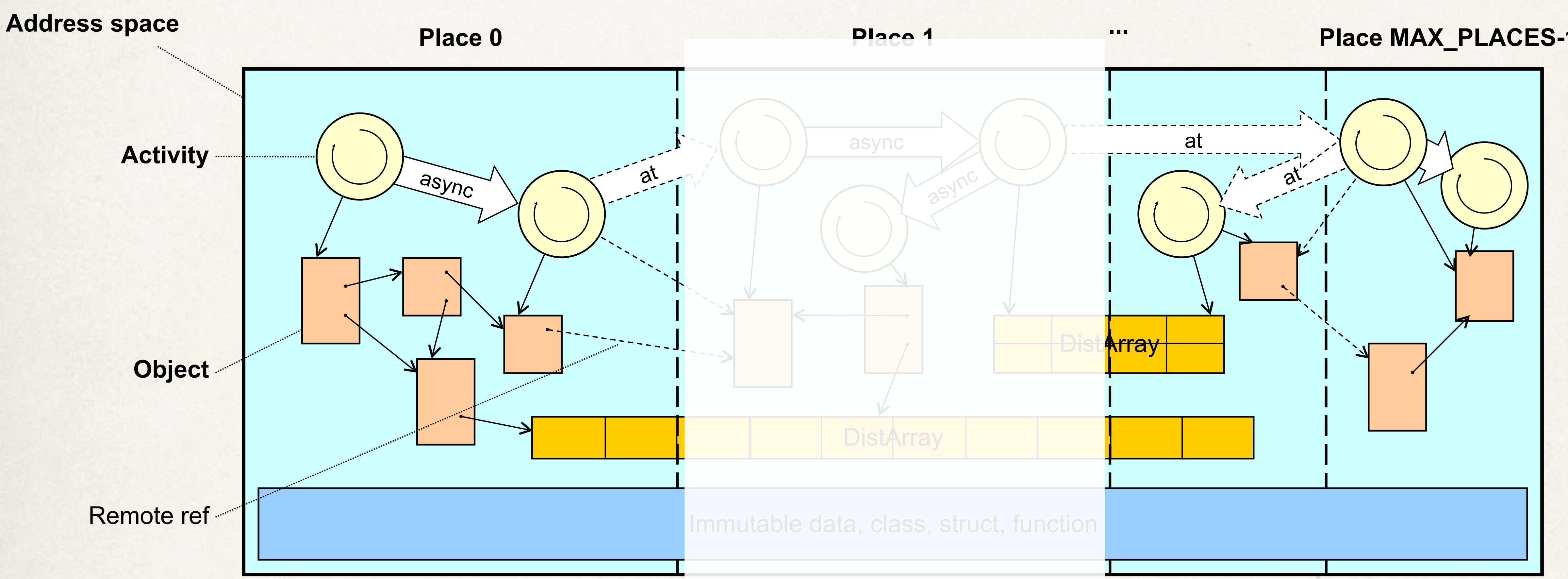
```
%X10_NPLACES=4
Hello from place 1
Hello from place 2
Hello from place 0
Hello from place 3
Hello from everywhere
```

Async Partitioned Global Address Space



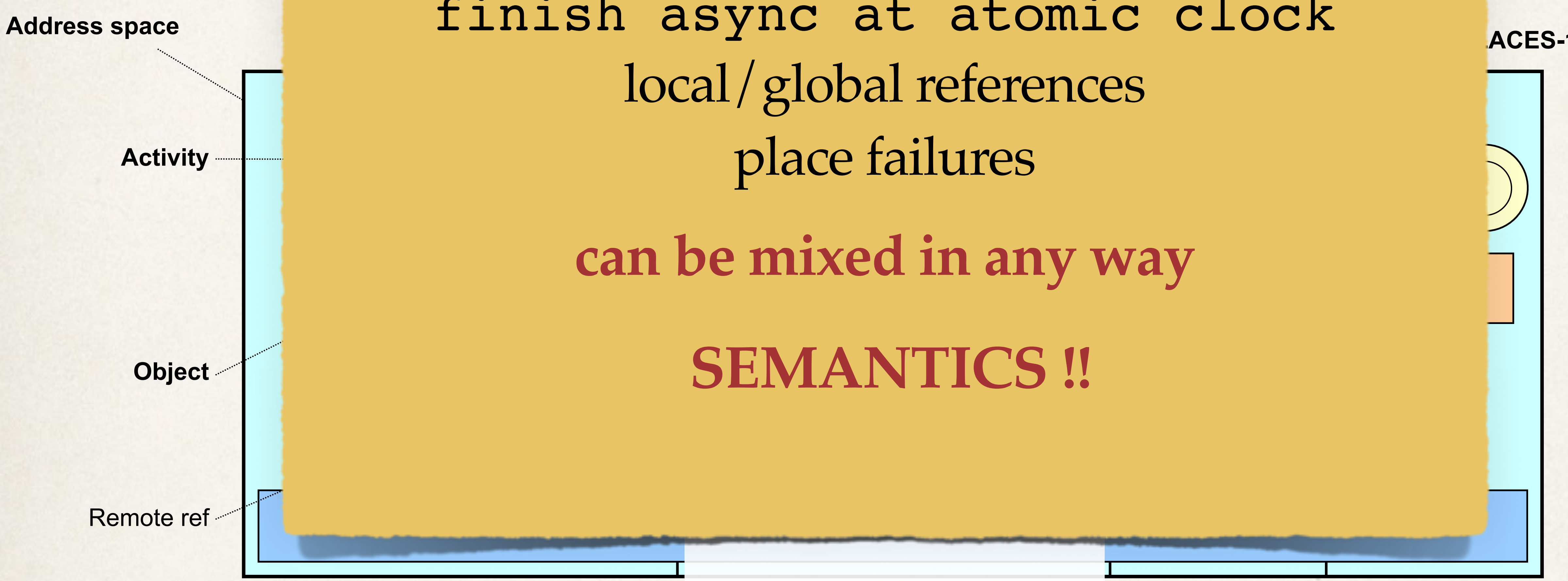
- A global address space is divided into multiple *places* (=computing nodes)
  - Each place can contain *activities* and *objects*
- An object belongs to a specific place, but can be *remotely referenced*
- **DistArray** is a data structure whose elements are scattered over multiple places

Resilient X10: if a node fails….



- it is relatively easy to **localize the impact of place death**
  - Objects in other places are still alive, but remote references become inaccessible
  - Execution continues using the remaining nodes
  - Happens Before Relation between remaining statements is preserved (HB Invariance) – no new race conditions, or sequentialization induced by failure.

Resilie



`finish async at atomic clock`

local/global references

place failures

can be mixed in any way

SEMANTICS !!

- it is relatively easy to **localize the impact of place death**
  - Objects in other places are still alive, but remote references become inaccessible
  - Execution continues using the remaining nodes
  - Happens Before Relation between remaining statements is preserved (HB Invariance) – no new race conditions, or sequentialization induced by failure.

TX10

Semantics of (Resilient) X10 [ECOOP 2014]
S.Crafa, D.Cunningham, V.Saraswat, A.Shinnar, O.Tardieu

object id → o; global object id → $o\$p$; exception → E | DPE

$$\begin{array}{llll}
\textit{Values} & v & ::= & o \mid o\$p \mid \mathsf{E} \mid \mathsf{DPE} \\
\textit{Expressions} & e & ::= & v \mid x \mid e.f \mid \{f{:}e, \ldots, f{:}e\} \mid \texttt{globalref}\ e \mid \texttt{valof}\ e \\
\textit{Statements} & s & ::= & \texttt{skip;} \mid \texttt{throw}\ v \mid \texttt{val}\ x = e\ s \mid e.f = e; \mid \{s\ \ t\} \\
 & & & \texttt{at}(p)\texttt{val}\ x = e\ \texttt{in}\ s \mid \texttt{async}\ s \mid \texttt{finish}\ s \mid \texttt{try}\ s\ \texttt{catch}\ t \\
 & & & \overline{\texttt{at}(p)\ s} \mid \overline{\texttt{async}\ s} \mid \texttt{finish}_{\mu}\ s \\
\textit{Configurations} & k & ::= & \langle s, g \rangle \mid g
\end{array}$$

error propagation and handling (→ $\texttt{try}\ s\ \texttt{catch}\ t$)

$$\textit{Global heap}\ \ g ::= \emptyset \mid g \cdot [p \mapsto h] \qquad \textit{Local heap}\ \ h ::= \emptyset \mid h \cdot [o \mapsto (\tilde{f}_i : \tilde{v}_i)]$$

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g \rangle \longrightarrow_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{\mathsf{E}\times}_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{\mathsf{E}\otimes}_p \langle s', g' \rangle \mid g'$$

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g \rangle \longrightarrow_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{E\times}_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{E\otimes}_p \langle s', g' \rangle \mid g'$$

Async failures arise in parallel threads
and are caught by the inner `finish` waiting for their termination

```
finish {async throw E   async s2}
```

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g \rangle \longrightarrow_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{\mathsf{E}\times}_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{\mathsf{E}\otimes}_p \langle s', g' \rangle \mid g'$$

Async failures arise in parallel threads
and are caught by the inner `finish` waiting for their termination

```
finish {async throw E   async s2}
```

Synch failures lead to the failure of any sync continuation
leaving async (remote) running code free to terminate

```
{async at(p)s1   throw E   s2}
```

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g\rangle \longrightarrow_p \langle s', g'\rangle \mid g' \qquad \langle s, g\rangle \xrightarrow{E\times}_p \langle s', g'\rangle \mid g' \qquad \langle s, g\rangle \xrightarrow{E\otimes}_p \langle s', g'\rangle \mid g'$$

Proved in Coq

Async failures arise in parallel threads
and are caught by the inner `finish` waiting for their termination

```
finish {async throw E   async s2}
```

Proved in Coq

Synch failures lead to the failure of any sync continuation
leaving async (remote) running code free to terminate

```
{async at(p)s1   throw E   s2}
```

Semantics of (Resilient) X10

- Small-step transition system, mechanised in Coq
- **non in ChemicalAM style** (better fits the centralised view of the distributed program)

$$\langle s, g \rangle \longrightarrow_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{\mathsf{E}\times}_p \langle s', g' \rangle \mid g' \qquad \langle s, g \rangle \xrightarrow{\mathsf{E}\otimes}_p \langle s', g' \rangle \mid g'$$

Proved in Coq

Absence of stuck states

(the proof can be run, yielding an interpreter for TX10)

Semantics of Resilient X10

smoothly scales to node failure, with

- global heap is a partial map: ***dom(g)* collects non failed places**
- executing a statement at failed place results in a DPE
- place shift at failed place results in a DPE
- remote exceptions flow back at the remaining finish **masked** as DPE

contextual rules modified accordingly

(Place Failure)

$$\frac{p \in dom(g)}{\langle s, g\rangle \longrightarrow_p \langle s, g \setminus \{(p, g(p)\}\rangle}$$

$$\frac{p \notin dom(g)}{\begin{array}{r}\langle \mathtt{skip}, g\rangle \xrightarrow{\mathsf{DPE}\otimes}_p g \\ \langle \mathtt{async}\, s, g\rangle \xrightarrow{\mathsf{DPE}\otimes}_p g \\ \langle \mathtt{at}(p)\, s, g\rangle \xrightarrow{\mathsf{DPE}\otimes}_q g\end{array}}$$

Semantics of Resilient X10

- **Happens Before Invariance**
  - failure of place q does not alter the happens before relationship between statement instances at places other than q

$\overline{\texttt{at}}(0)\,\{\,\overline{\texttt{at}}(p)\ \texttt{finish}\ \overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\quad s_2\}$

s2 runs at 0 <u>after</u> s1

$\overline{\texttt{at}}(0)\,\texttt{finish}\,\{\,\overline{\texttt{at}}(p)\{\overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\}\quad s_2\}$

s2 runs at 0 in <u>parallel</u> with s1

Semantics of Resilient X10

- **Happens Before Invariance**
  - failure of place q does not alter the happens before relationship between statement instances at places other than q

$\overline{\texttt{at}}(0)\,\{\,\overline{\texttt{at}}(p)\ \texttt{finish}\ \overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\quad s_2\}$

s2 runs at 0 <u>after</u> s1

p fails while s1 is running at q

$\overline{\texttt{at}}(0)\,\texttt{finish}\,\{\,\overline{\texttt{at}}(p)\{\overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\}\quad s_2\}$

s2 runs at 0 in <u>parallel</u> with s1

Semantics of Resilient X10

- **Happens Before Invariance**
  - failure of place q does not alter the happens before relationship between statement instances at places other than q

$\overline{\texttt{at}}(0)\,\{\,\overline{\texttt{at}}(\cancel{p})\ \texttt{finish}\ \overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1 \quad s_2\}$

s2 runs at 0 <u>after</u> s1

p fails while s1 is running at q

same behaviour!

$\overline{\texttt{at}}(0)\,\texttt{finish}\,\{\,\overline{\texttt{at}}(\cancel{p})\{\overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\} \quad s_2\}$

s2 runs at 0 in <u>parallel</u> with s1

Semantics of Resilient X10

- **Happens Before Invariance**
  - failure of place q does not alter the happens before relationship between statement instances at places other than q

$\mathsf{DPE}\otimes$ ***flows at place 0 discarding s1***

$$\overline{\texttt{at}}(0)\,\{\,\overline{\texttt{at}}(p)\ \texttt{finish}\ \overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\quad s_2\}$$

throws v

$$\overline{\texttt{at}}(0)\,\texttt{finish}\,\{\,\overline{\texttt{at}}(p)\{\overline{\texttt{at}}(q)\ \overline{\texttt{async}}\,s_1\}\quad s_2\}$$

$v\times$ ***flows at place 0 while s2 is running***

Equational theory for (Resilient) X10

$\langle s, g \rangle \;\cong\; \langle t, g \rangle$ **equivalent configurations** when

- transition steps are weakly bi-simulated
- ***under any modification of the shared heap by current activities***
  (object field update, object creation, place failure)

Equational theory for (Resilient) X10

$\langle s, g \rangle \;\cong\; \langle t, g \rangle$ **equivalent configurations** when

- transition steps are weakly bi-simulated
- ***under any modification of the shared heap by current activities***
  (object field update, object creation, place failure)

$\langle s, g \rangle \; \mathcal{R} \; \langle t, g \rangle \;\; \textit{whenever}$

1. $\vdash \mathsf{isSync}\, s \;\; \textit{iff} \;\; \vdash \mathsf{isSync}\, t$
2. $\forall p, \forall \Phi \;\; \textit{environment move}$

$$\textit{if} \;\; \langle s, \Phi(g) \rangle \xrightarrow{\lambda}_p \langle s', g' \rangle \;\; \textit{then} \;\; \exists t'.\; \langle t, \Phi(g) \rangle \stackrel{\lambda}{\Longrightarrow}_p \langle t', g' \rangle$$

$$\textit{with} \;\; \langle s', g' \rangle \; \mathcal{R} \; \langle t', g' \rangle \;\; \textit{and} \;\; \textit{viceversa}$$

Equational theory for (Resilient) X10

$\langle s, g \rangle \;\cong\; \langle t, g \rangle$ **equivalent configurations** when

- transition steps are weakly bi-simulated
- ***under any modification of the shared heap by current activities***
  (object field update, object creation, place failure)

$\langle s, g \rangle \; \mathcal{R} \; \langle t, g \rangle$ *whenever*

1. $\vdash \mathsf{isSync}\, s$ *iff* $\vdash \mathsf{isSync}\, t$
2. $\forall p, \forall \Phi$ *environment move*

$$\textit{if } \langle s, \Phi(g) \rangle \xrightarrow{\lambda}_p \langle s', g' \rangle \textit{ then } \exists t'.\ \langle t, \Phi(g) \rangle \overset{\lambda}{\Longrightarrow}_p \langle t', g' \rangle$$

$$\textit{with } \langle s', g' \rangle \; \mathcal{R} \; \langle t', g' \rangle \textit{ and viceversa}$$

models the update of g:

$$dom(\Phi(g)) = dom(g) \textit{ and}$$
$$\forall p {\in} dom(g)\ dom(g(p)) \subseteq dom(\Phi(g)(p))$$

Equational theory for (Resilient) X10

$\langle s, g \rangle \ \cong \ \langle t, g \rangle$ **equivalent configurations** when

- transition steps are weakly bi-simulated
- ***under any modification of the shared heap by current activities***
  (object field update, object creation, place failure)

$\langle s, g \rangle \ \mathcal{R} \ \langle t, g \rangle$ *whenever*

1. $\vdash$isSync s *iff* $\vdash$isSync t
2. $\forall p, \forall \Phi$ *environment move*

   $$\textit{if} \ \langle s, \Phi(g) \rangle \xrightarrow{\lambda}_p \langle s', g' \rangle \ \textit{then} \ \exists t'. \ \langle t, \Phi(g) \rangle \overset{\lambda}{\Longrightarrow}_p \langle t', g' \rangle$$

   $$\textit{with} \ \langle s', g' \rangle \ \mathcal{R} \ \langle t', g' \rangle \ \textit{and viceversa}$$

models the update of g:

$$dom(\Phi(g)) = dom(g) \ \textit{and}$$
$$\forall p \in dom(g) \ dom(g(p)) \subseteq dom(\Phi(g)(p))$$

Bisimulation whose Bisimilarity is a congruence

Equational theory for (Resilient) X10

$$\{\{s \quad t\} \quad u\} \cong \{s \quad \{t \quad u\}\}$$

$$\vdash \textsf{isAsync}\ s \qquad \texttt{try}\,\{s\ t\}\,\texttt{catch}\,u \cong \{\texttt{try}\,s\,\texttt{catch}\,u \quad \texttt{try}\,t\,\texttt{catch}\,u\}$$

$$\texttt{at}(p)\{s\ t\} \not\cong_{\mathrm{R}} \{\texttt{at}(p)s \quad \texttt{at}(p)t\}$$

$$\texttt{at}(p)\texttt{at}(q)s \not\cong_{\mathrm{R}} \texttt{at}(q)s$$

$$\texttt{async at}(p)s \not\cong_{\mathrm{R}} \texttt{at}(p)\,\texttt{async}\,s$$

if s throws a sync exc. and home is failed, then l.h.s. throws a masked DPEx while r.h.s. re-throws vx since synch exc are not masked by DPE

$$\texttt{finish}\,\{s \quad t\} \cong \texttt{finish}\,s \quad \texttt{finish}\,t$$

$$\texttt{finish}\,\{s \quad \texttt{async}\,t\} \cong \texttt{finish}\,\{s \quad t\}$$

$$\texttt{finish at}(p)\,s \not\cong_{\mathrm{R}} \texttt{at}(p)\,\texttt{finish}\,s$$

Conclusions

- **Concurrecy is critical for Programming Languages**
  - heterogeneous concurrency models (Distribution)

- **What is the right level of abstraction?**
  - What are good abstractions? Expressive, flexible, easy to reason about, easy to implement in a scalable/resilient way

- **Formal method to experiment!**
  - test new primitive, *new mix* of primitives
  - tool to reason about programs

(Par Left)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda{=}\epsilon, v\times & \langle\{s\ \ t\}, g\rangle \xrightarrow{\lambda}_p \langle\{s'\ \ t\}, g'\rangle \mid \langle t, g'\rangle \\ \lambda = v\otimes & \langle\{s\ \ t\}, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g' \end{array}}$$

(Par Right)

$$\frac{\vdash \mathsf{isAsync}\, t \qquad \langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\langle\{t\ \ s\}, g\rangle \xrightarrow{\lambda}_p \langle\{t\ \ s'\}, g'\rangle \mid \langle t, g'\rangle}$$

(Place Shift)

$$\frac{(v', g') = \mathsf{copy}(v, q, g)}{\langle \mathtt{at}(q)\mathtt{val}\, x = v\, \mathtt{in}\, s,\ g\rangle \longrightarrow_p \langle \overline{\mathtt{at}}(q)\{s[^{v'}/_x]\ \mathtt{skip}\}, g'\rangle}$$

(At)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_q \langle s', g'\rangle \mid g'}{\langle \overline{\mathtt{at}}(q)\, s, g\rangle \xrightarrow{\lambda}_p \langle \overline{\mathtt{at}}(q)\, s', g'\rangle \mid g'}$$

(Spawn)

$$\frac{}{\langle \texttt{async}\, s, g\rangle \longrightarrow_p \langle \overline{\texttt{async}}\, s, g\rangle}$$

(Async)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda = \epsilon & \langle \overline{\texttt{async}}\, s, g\rangle \xrightarrow{\lambda}_p \langle \overline{\texttt{async}}\, s', g'\rangle \mid g' \\ \lambda{=}v\times, v\otimes & \langle \overline{\texttt{async}}\, s, g\rangle \xrightarrow{v\times}_p \langle \overline{\texttt{async}}\, s', g'\rangle \mid g' \end{array}}$$

(Finish)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle}{\langle \texttt{finish}_\mu\, s, g\rangle \longrightarrow_p \langle \texttt{finish}_{\mu\cup\lambda}\, s', g'\rangle}$$

(End Finish)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p g' \quad \lambda' = \mathsf{E}\otimes \;\; \mathit{if}\, \lambda\cup\mu\neq\emptyset \;\mathit{else}\; \epsilon}{\langle \texttt{finish}_\mu\, s, g\rangle \xrightarrow{\lambda'}_p g'}$$

(Exception)

$$\frac{}{\langle \texttt{throw}\, v, g\rangle \xrightarrow{v\otimes}_p g}$$

(Try)

$$\frac{\langle s, g\rangle \xrightarrow{\lambda}_p \langle s', g'\rangle \mid g'}{\begin{array}{ll} \lambda{=}\epsilon, v\times & \langle \texttt{try}\, s\, \texttt{catch}\, t, g\rangle \xrightarrow{\lambda}_p \langle \texttt{try}\, s'\, \texttt{catch}\, t, g'\rangle \mid g' \\ \lambda{=}v\otimes & \langle \texttt{try}\, s\, \texttt{catch}\, t, g\rangle \longrightarrow_p \langle \{s' \quad t\}, g'\rangle \mid \langle t, g'\rangle \end{array}}$$

(Skip)

$$\frac{}{\langle \texttt{skip}, g\rangle \longrightarrow_p g}$$

Plus rules for expression evaluation