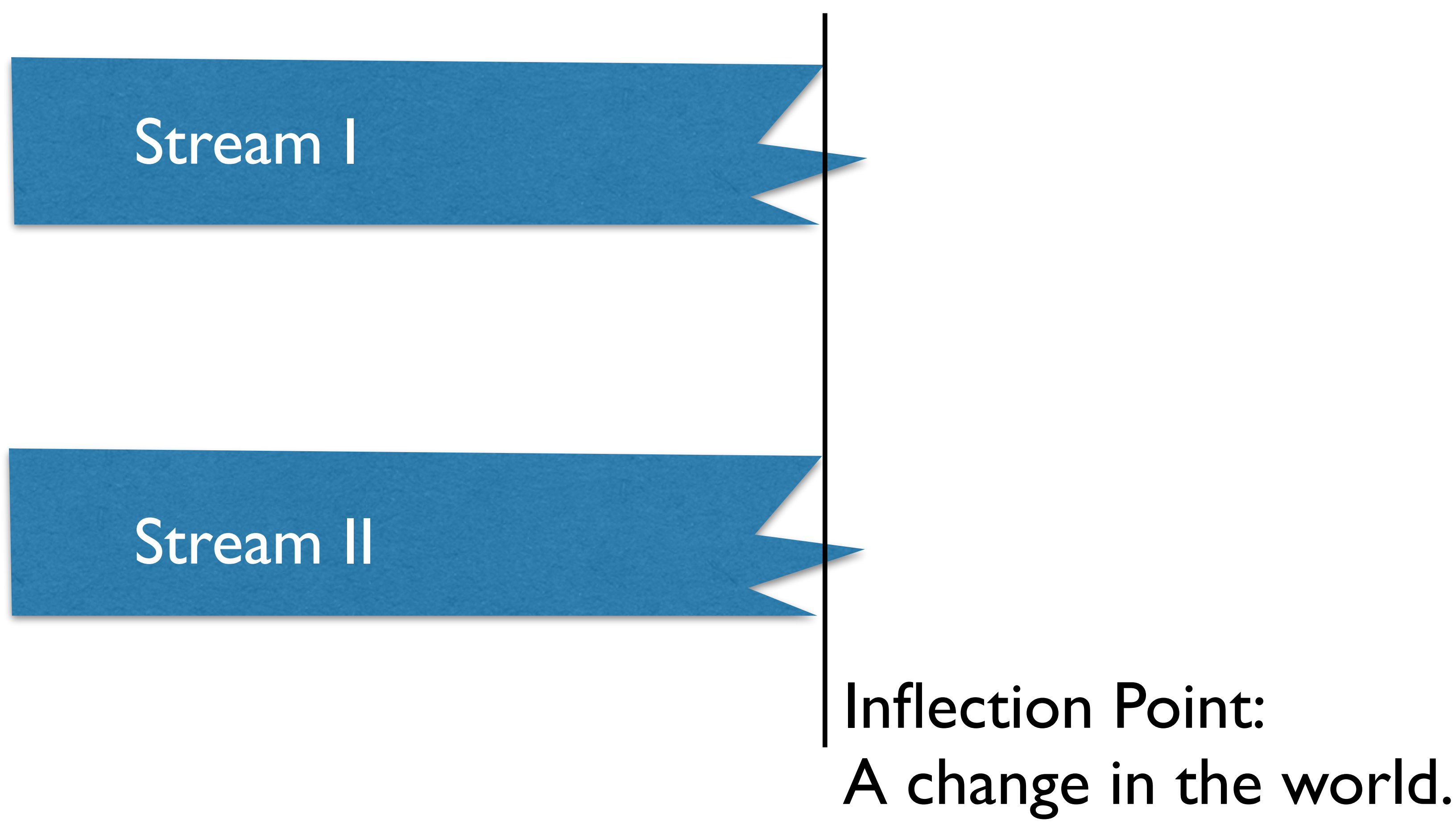
Confluences in Programming Languages Research

David Walker
Princeton University

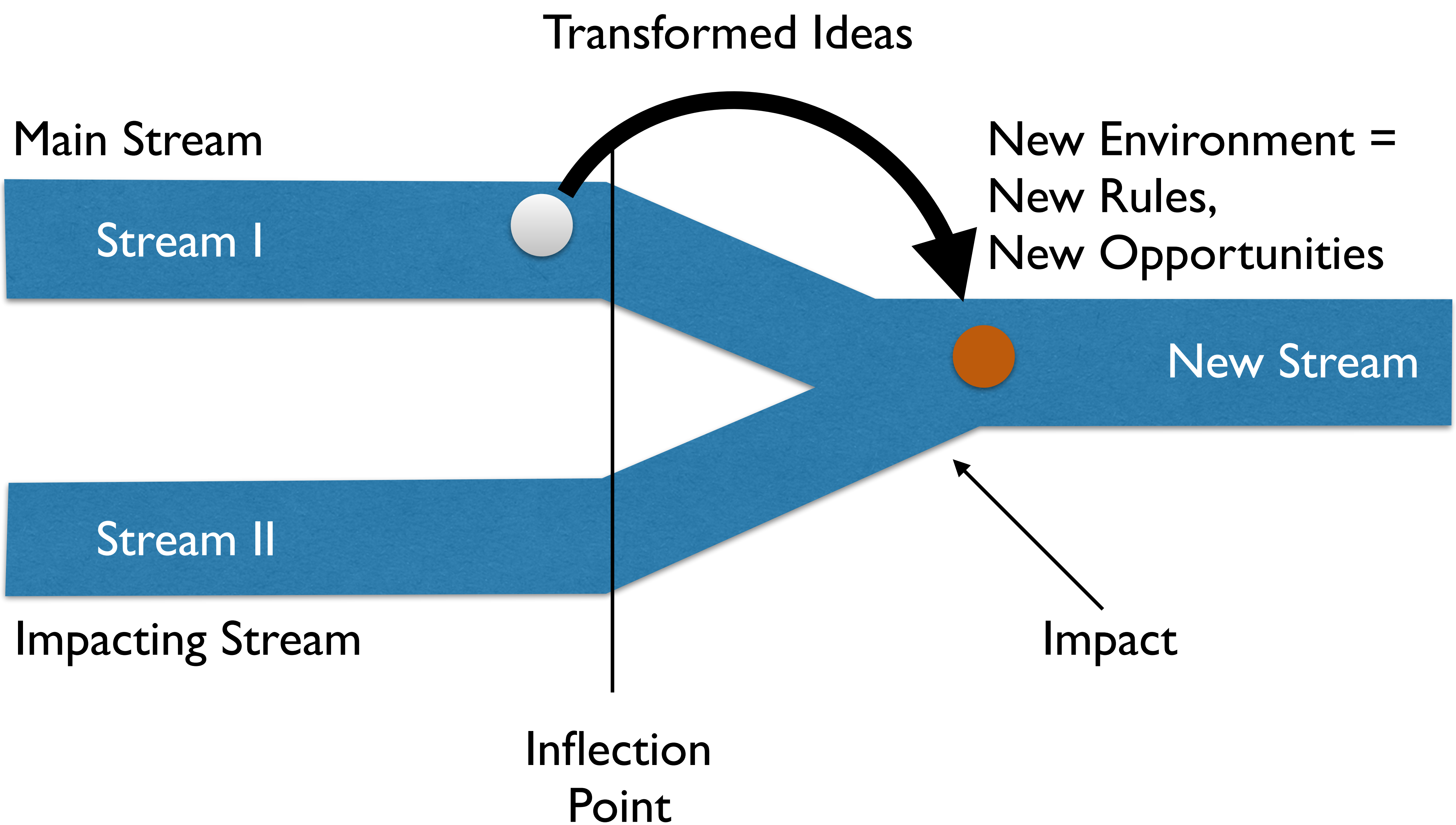
A confluence: The junction of two rivers

A Confluence of Ideas



Life in the Fast Lane: Viewed from the Confluence Lens. George Varghese, SIGCOMM CCR, 2015.

A Confluence of Ideas



Life in the Fast Lane: Viewed from the Confluence Lens. George Varghese, SIGCOMM CCR, 2015.

Impressionism as Confluence

Invisible strokes
and a focus on
realistic detail

Realism (1800s)

Photography
“Press a button, we do the rest” (Kodak 1888)

Life in the Fast Lane: Viewed from the Confluence Lens. George Varghese, SIGCOMM CCR, 2015.

Impressionism as Confluence

Invisible strokes
and a focus on
realistic detail

Realism (1800s)

Psychology

Freud (1836-1934):
emotion affects perception

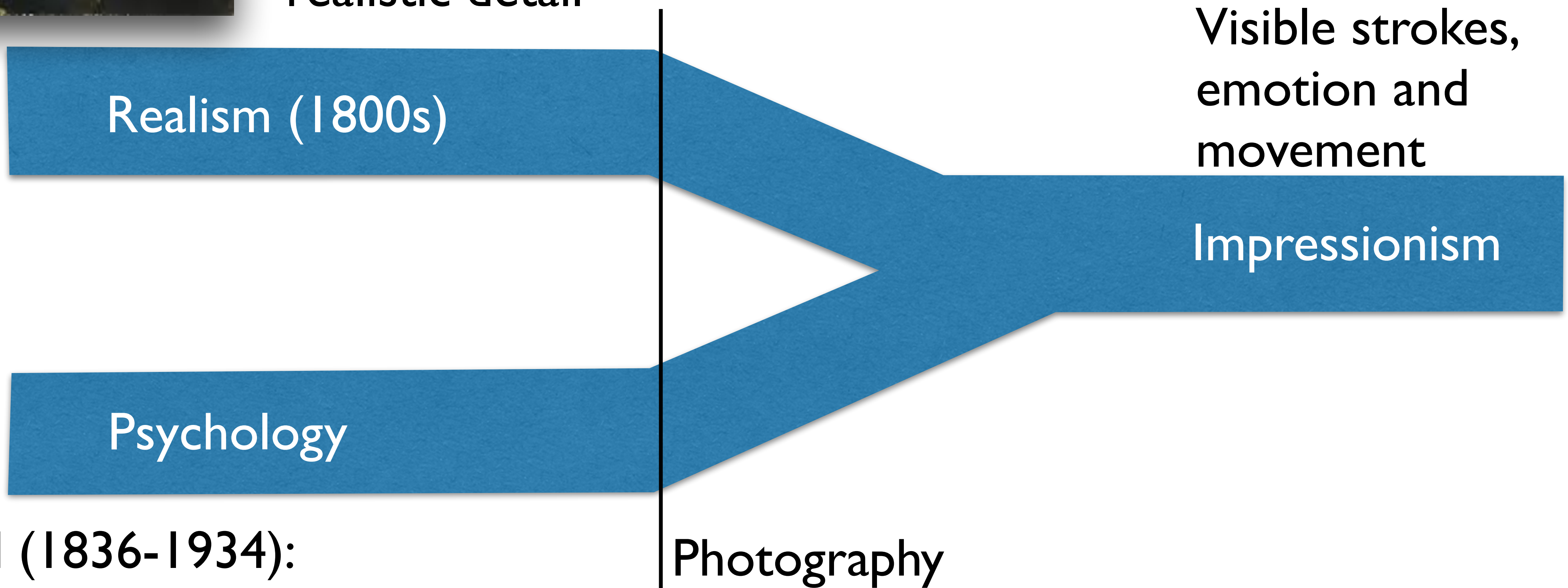
Photography
“Press a button, we do the rest” (Kodak 1888)

Life in the Fast Lane: Viewed from the Confluence Lens. George Varghese, SIGCOMM CCR, 2015.

Impressionism as Confluence

Invisible strokes
and a focus on
realistic detail

Visible strokes,
emotion and
movement



Freud (1836-1934):
emotion affects perception

Photography
"Press a button, we do the rest" (Kodak 1888)

Life in the Fast Lane: Viewed from the Confluence Lens. George Varghese, SIGCOMM CCR, 2015.

Networked Vehicles as Confluence

Threat model:
"watch out for that horse"

Motorized Vehicles

Networked Vehicles as Confluence

Threat model:
"watch out for
that horse"

Motorized
Vehicles

cars go
online

Networked Vehicles as Confluence

Threat model:
“watch out for that horse”

New threat models:
Hackers remotely take control of Jeep on highway

Motorized Vehicles

cars go online

Networked Vehicles as Confluence

Threat model:
"watch out for
that horse"

New threat models:
Hackers remotely take
control of Jeep on highway

Motorized Vehicles

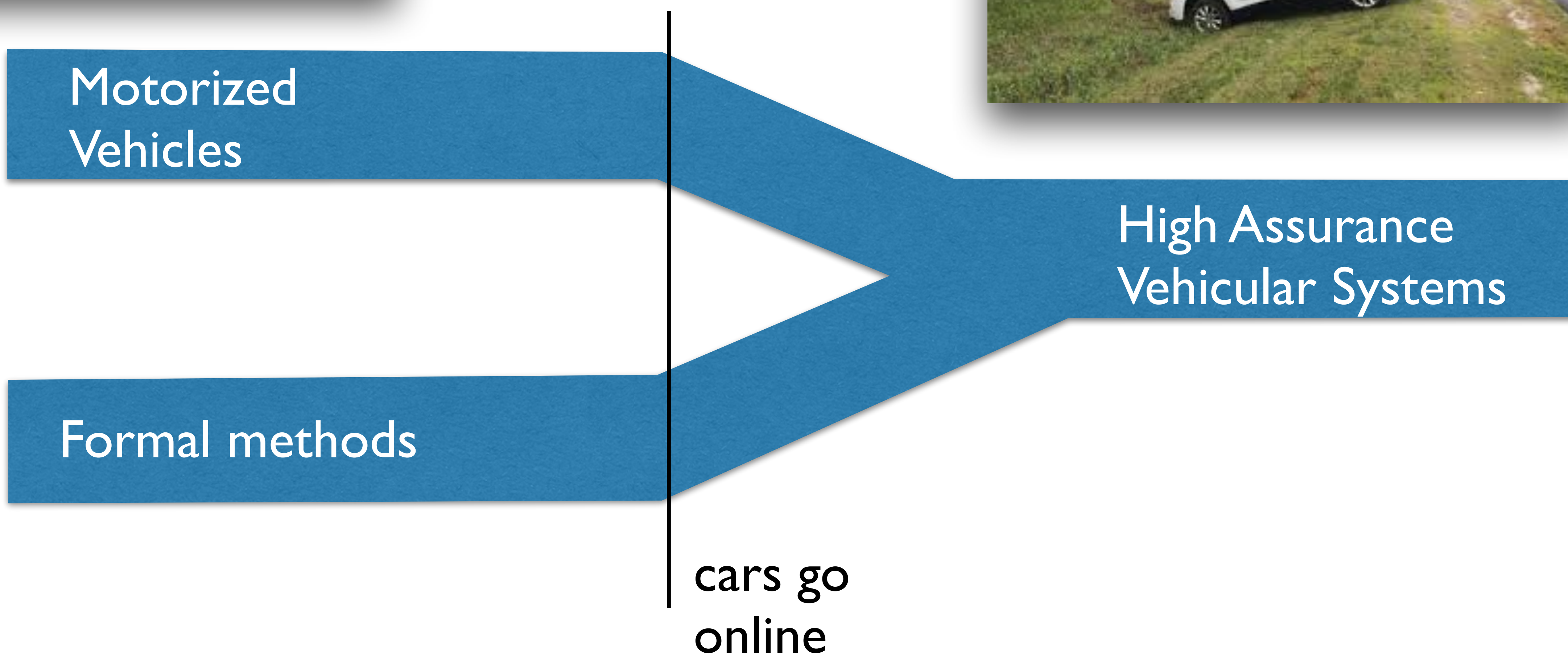
Formal methods

cars go online

Networked Vehicles as Confluence

Threat model:
“watch out for that horse”

New threat models:
Hackers remotely take control of Jeep on highway



Networked Vehicles as Confluence

Threat model:
"watch out for
that horse"

New threat models:
Hackers remotely take
control of Jeep on highway

Motorized Vehicles

Application Domain

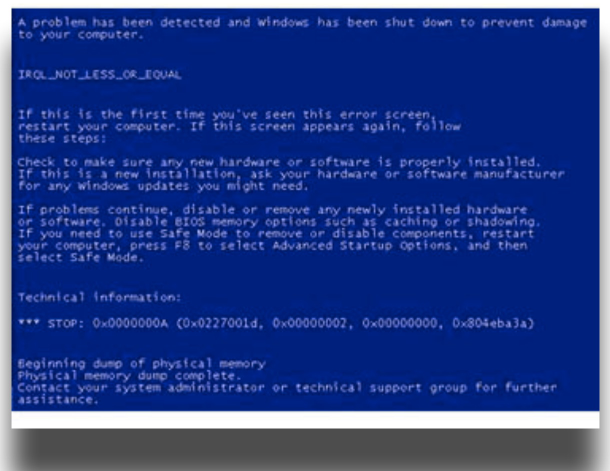
Formal methods

Basic PL Research

cars go online

High Assurance Vehicular Systems

Operating System Reliability as Confluence



Blue Screen of Death

Testing is hopelessly incomplete

Operating System Reliability

Model Checking

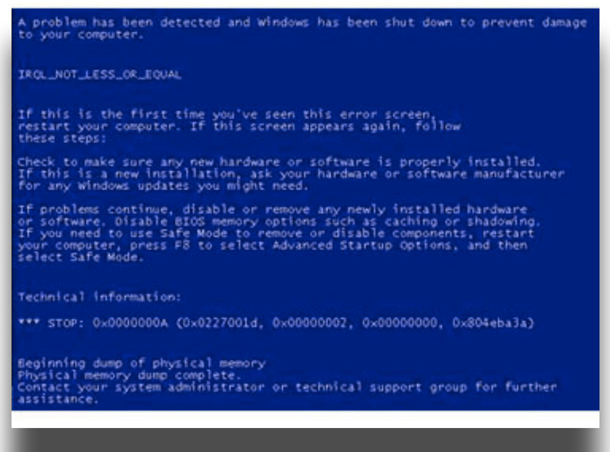
Basic research breakthroughs:

- data structures
- algorithms
- abstraction

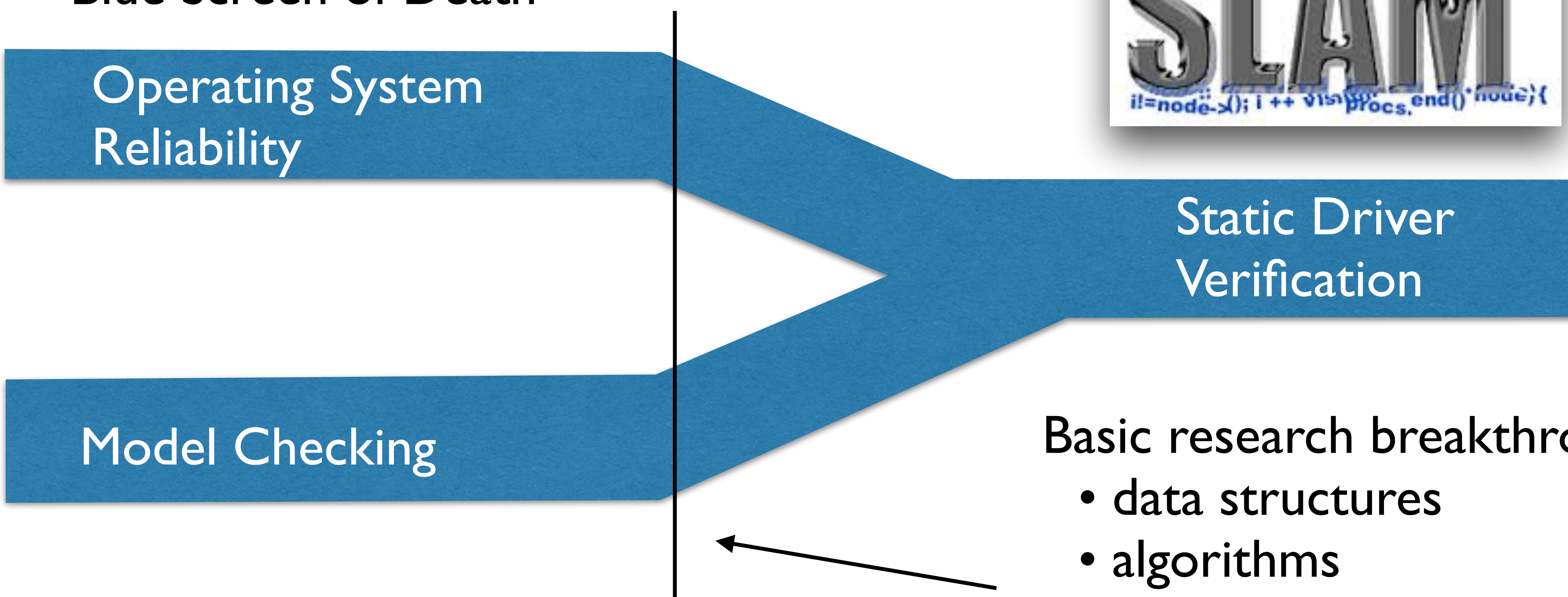
Hardware:

- thanks Intel!

Operating System Reliability as Confluence



Blue Screen of Death



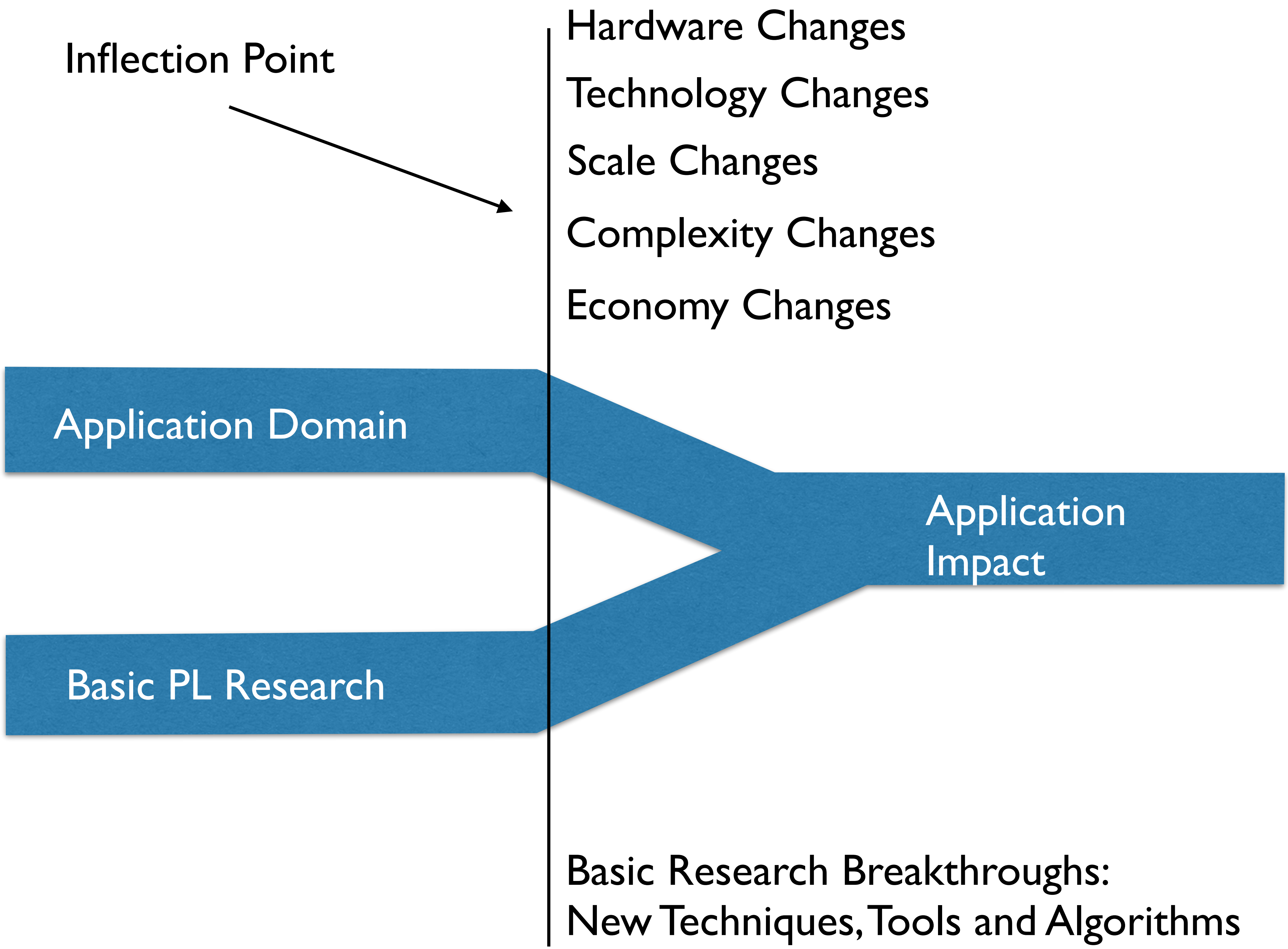
Basic research breakthroughs:

- data structures
- algorithms
- abstraction

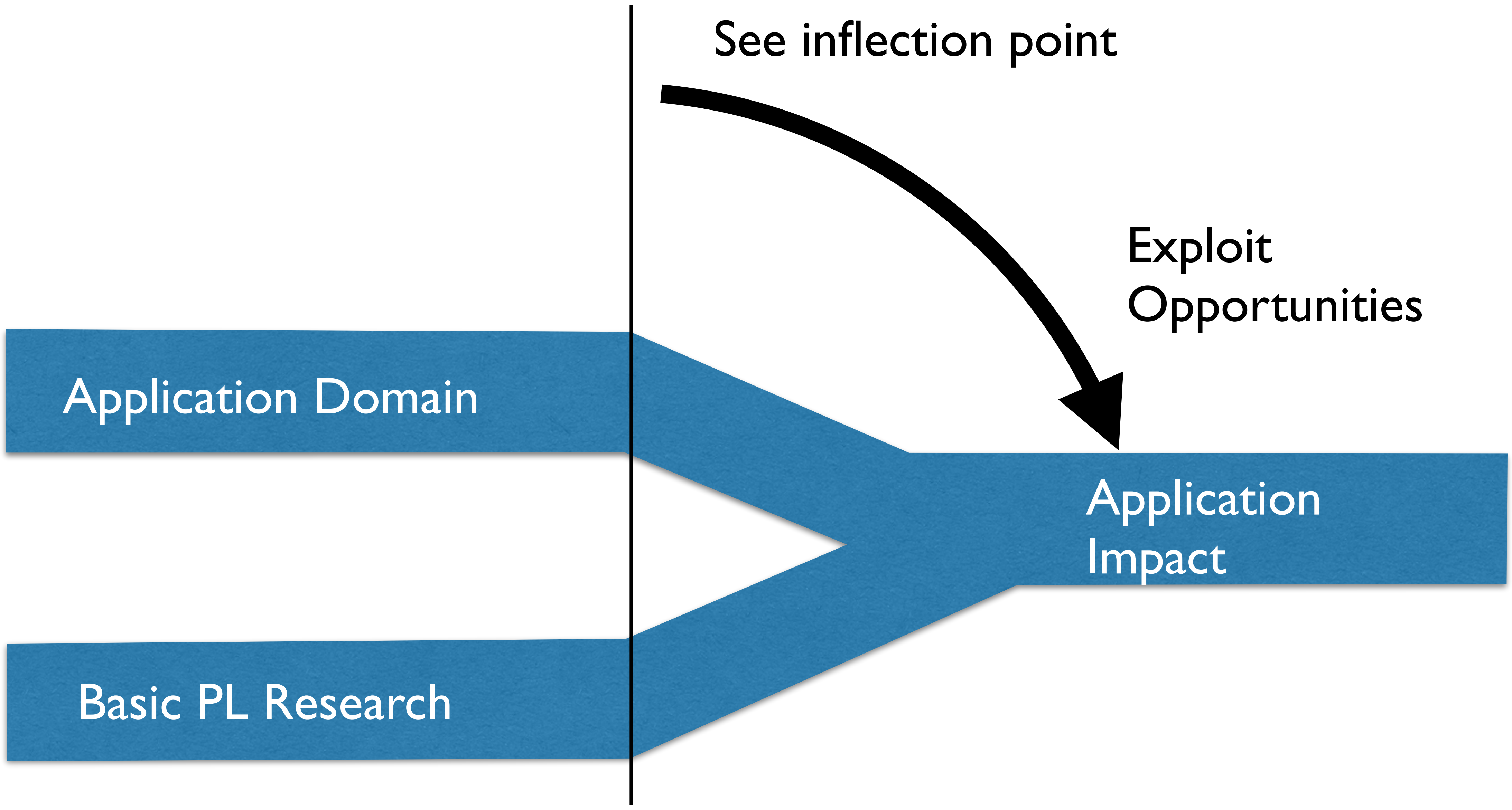
Hardware:

- thanks Intel!

Confluences in Programming Languages Research



Why Confluences?



Inflection points separate fads from opportunity for real change.

Early access maximizes influence on thought leaders.

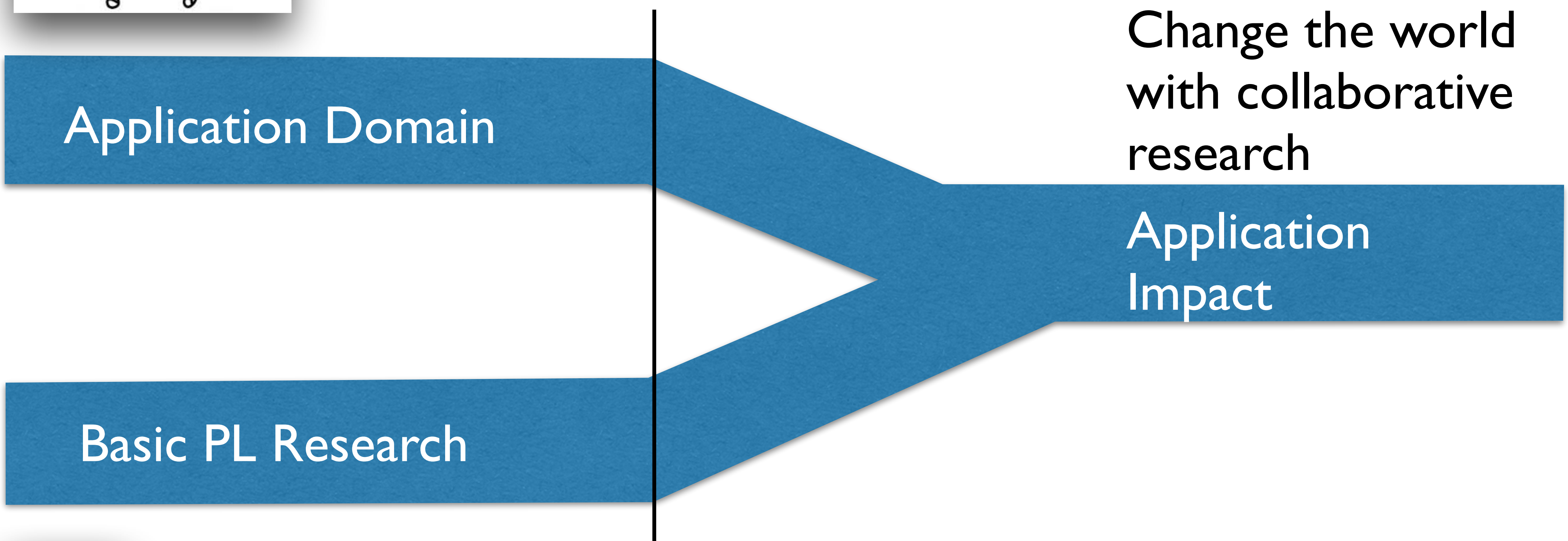
How Confluences?

Can we really see these inflection points as they happen?

Not always! *We* (often) can't!

We can make friends.
Perhaps *they* can

Change the world
with collaborative
research



Deep, general, reusable,
hard-to-learn skills

Two Confluences In My Career
And What I Learned From Them

Grad School: Learning Skills

- Confluences in reliable systems implementation
- Inflection point: a breakthrough in basic research

Professor Life: Making Friends

- Confluences in network configuration
- Inflection point: growth of data centers & industrial networks

Grad School: Learning Skills
Confluences in Reliable Systems Implementation

With Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Fred Smith, Stephanie Weirich, Steve Zdancewic

Stream 1: Basic Research: Type Safety

An Ever-So-Brief History of Modern Type Safety Proofs

Dynamic Typing
in a Statically Typed Language*

Martín Abadi† Luca Cardelli† Benjamin Pierce‡ Gordon Plotkin§

Abstract

Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and generation of more efficient object code than languages where all type consistency checks are performed at run time. However, even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. To handle such situations safely, we propose to add a type `Dynamic` whose values are pairs of a value `v` and a type tag `T` where `v` has the type denoted by `T`. Instances of `Dynamic` are built with an explicit tagging construct and inspected with a type safe `typecase` construct.

This paper explores the syntax, operational semantics, and denotational semantics of a simple language including the type `Dynamic`. We give examples of how dynamically typed values can be used in programming. Then we discuss an operational semantics for our language and obtain a soundness theorem. We present two formulations of the denotational semantics of this language and relate them to the operational semantics. Finally, we consider the implications of polymorphism and some implementation issues.

Dynamic Typing in a Statically Typed Language.
Abadi, Cardelli, Pierce, Plotkin.

Semantic Domains:

V = B0 + B1 + … + F + W + D
F = V➞V
D = TypeCode x V
W = { . }

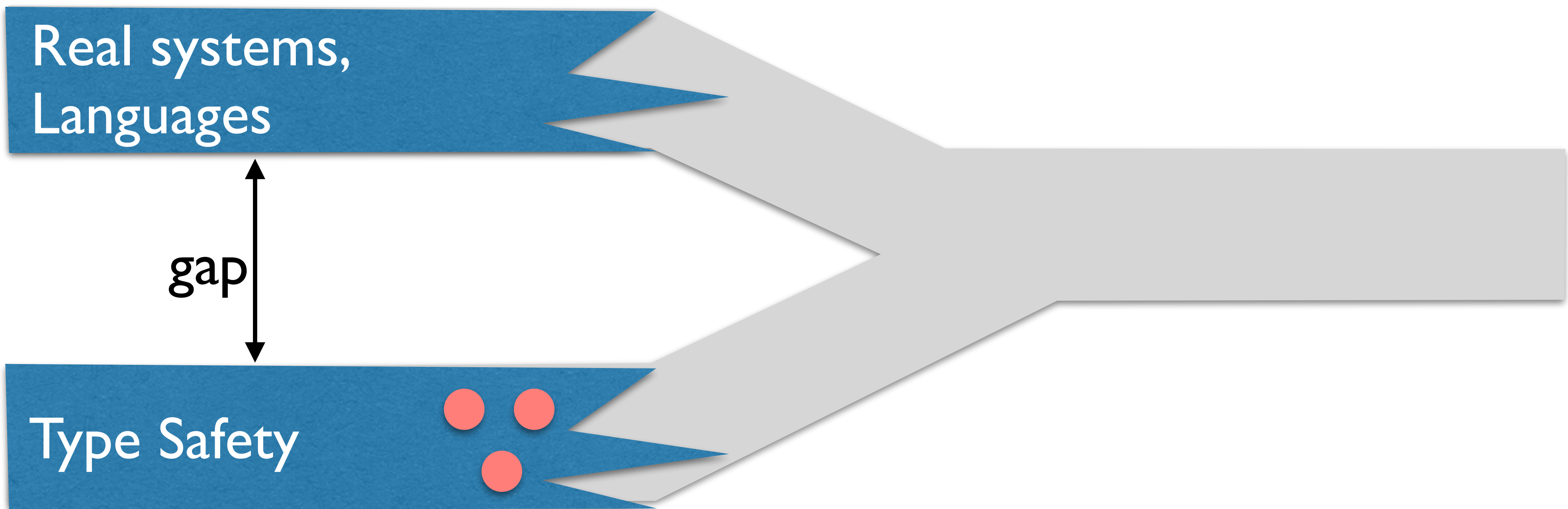
Proof:

Metric space argument shows the existence of the semantic relation.

semantic relation.

History from: *A Syntactic Approach to Type Soundness.* Wright and Felleisen. 1994

Where we were at:

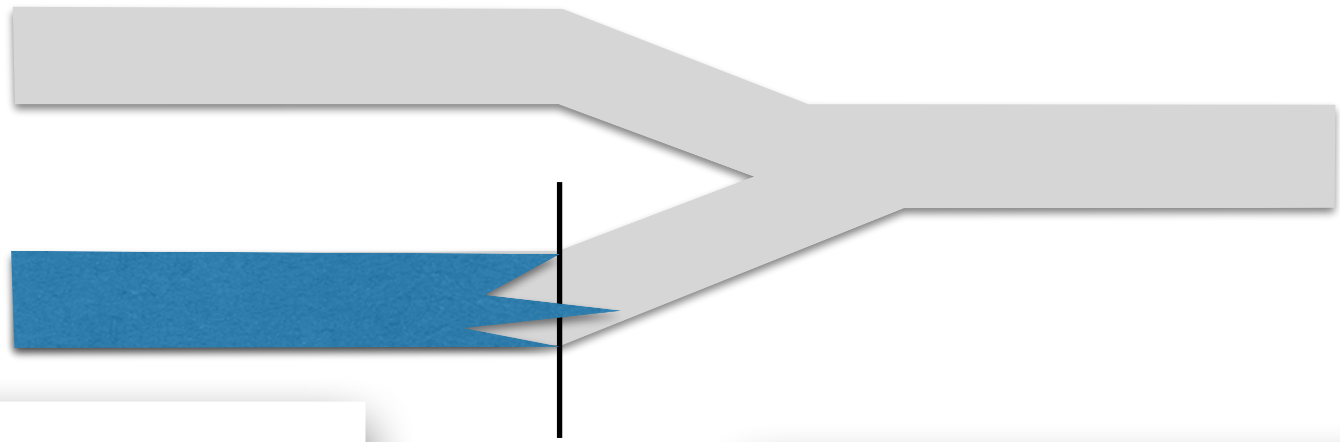


Milner Tofte MacQueen
Damas Mitchell
Plotkin Harper Felleisen
Lillibridge
Martin-Lof Kahn Leroy
Duba Talpin Gifford
Friedman … Many More …

Tiny, elegant languages

Hard proofs that are getting harder and that change with each new feature

The Inflection Point: Simple Syntactic Methods



INFORMATION AND COMPUTATION 115, 38–94 (1994)

A Syntactic Approach to Type Soundness

ANDREW K. WRIGHT AND MATTHIAS FELLEISEN*

Department of Computer Science, Rice University, Houston, Texas 77251-1892

We present a new approach to proving type soundness for Hindley/Milner-style polymorphic type systems. The keys to our approach are (1) an adaptation of subject reduction theorems from combinatory logic to programming languages, and (2) the use of rewriting techniques for the specification of the language semantics. The approach easily extends from polymorphic functional languages to imperative languages that provide references, exceptions, continuations, and similar features. We illustrate the technique with a type soundness theorem for the core of STANDARD ML, which includes the first type soundness proof for polymorphic exceptions and continuations. © 1994 Academic Press, Inc.

A Syntactic Approach to Type Soundness.
Wright, Felleisen. Info. & Comp, 1994.

Key contributions:

- Semantics by syntactic program rewriting
- Check program states are well-typed at each step
  - Modern Type Preservation
- Demonstrated *reuse* of the same technique on a variety of features and series of languages

A Simplified Account of Polymorphic References

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

Abstract

A proof of the soundness of Tofte's imperative type discipline with respect to a structured operational semantics is given. The presentation is based on a semantic formalism that combines the benefits of the approaches considered by Wright and Felleisen, and by Tofte, leading to a particularly simple proof of soundness of Tofte's type discipline.

Keywords: formal semantics, functional programming, programming languages, type theory, references and assignment.

A Simplified Account of Polymorphic References.
Robert Harper. Information Processing Letters 1994

. . . .

Modern Canonical Forms, Progress!
Harper, influenced by Martin-Löf, Plotkin

Confluences in Reliable Systems Implementation

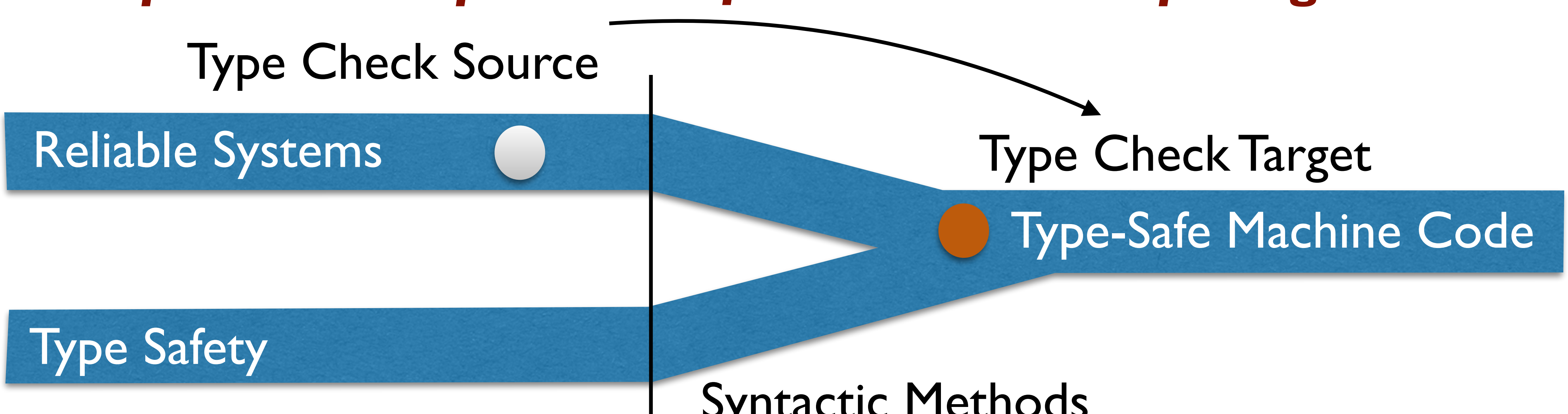
Type safety in practice:

- the foundation of mobile code security (Java & JVM)
- the foundation of promising systems architectures (SPIN OS)
- typed interfaces + type safety = secure, efficient sandboxes

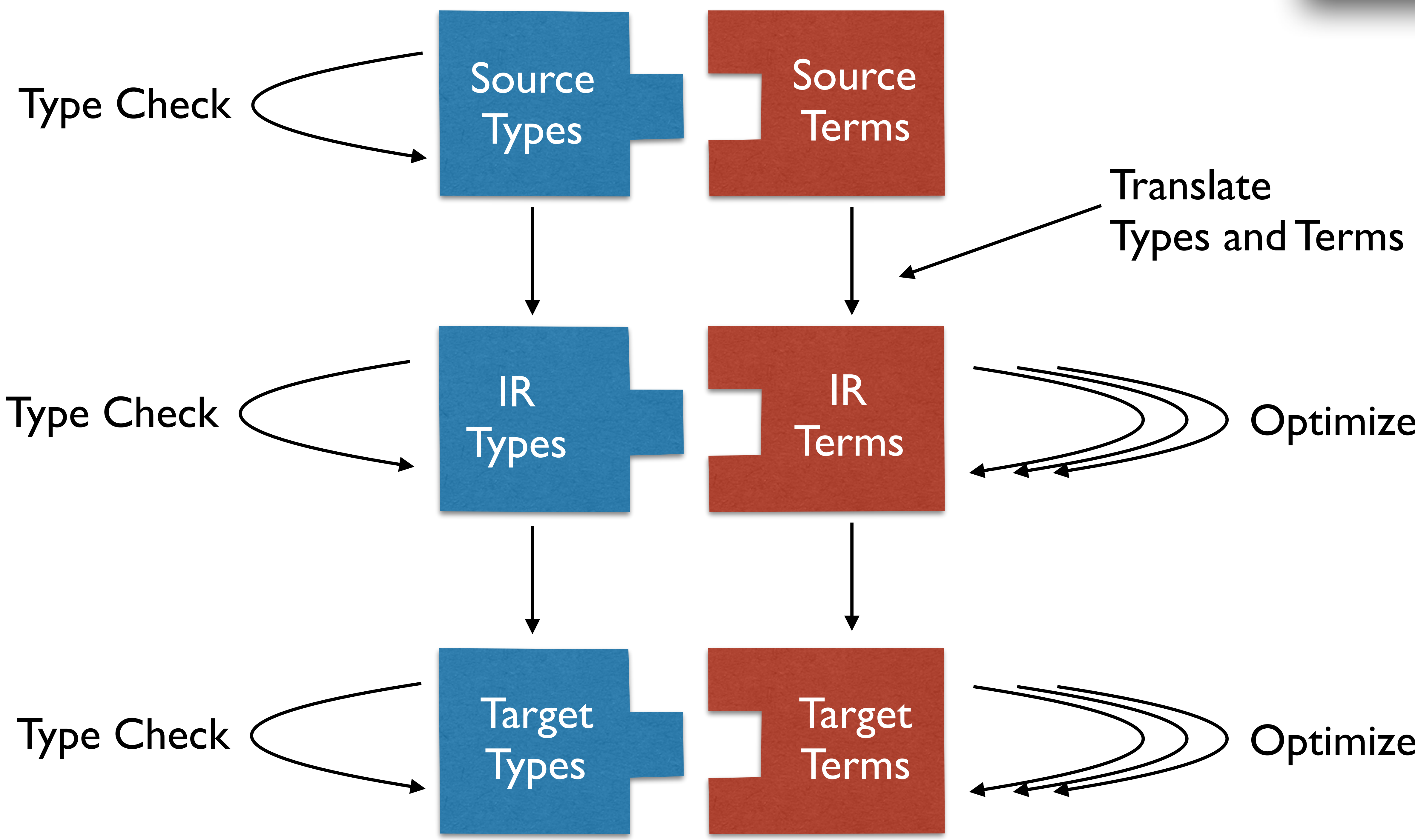
But type checking happened at the source

- consumers had to trust a compiler to preserve safety invariants
- compilers are 100s of thousands, millions LOC — errors inevitable

Can we pull the compiler out of the trusted computing base?



Typed Intermediate and Target Languages



TIL: A Type-directed, Optimizing Compiler for ML. Tarditi, Morrisett, Cheng, Stone, Harper, Lee. PLDI 96.
Safe Kernel Extensions without Run-time Checking. George Necula, Peter Lee. OSDI 96.
From System F to Typed Assembly Language. Morrisett, Walker, Crary, Glew. POPL 98.

TALx86

```
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax
```

```
sum:

    beq ebx, ecx    % if ebx=0, jump to ecx

    add eax, ebx    % eax := eax + ebx

    sub ebx, 1      % decrement counter

    jump sum        % iterate loop
```

Register files R:

$R ::= \{eax = v, ebx = v, \ldots\}$

Register file types Γ:

$\Gamma ::= \{eax : \tau, ebx : \tau, \ldots\}$

Machine value types:

$\tau ::= \text{int32}$
$\quad | \ \text{int64}$
$\quad | \ \text{float32}$
$\quad | \ \Gamma$ % code ptr
$\quad | \ \alpha$ % abstract type
$\quad | \ \forall\alpha.\tau$ % universal

TALx86

```
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax
```

sum's code type:

```
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
```

```
sum:

    beq ebx, ecx   % if ebx=0, jump to ecx

    add eax, ebx   % eax := eax + ebx

    sub ebx, 1     % decrement counter

    jump sum       % iterate loop
```

TALx86

```
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax
```

sum's code type:

```
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
```

```
sum:
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }

      beq ebx, ecx    % if ebx=0, jump to ecx

      add eax, ebx    % eax := eax + ebx

      sub ebx, 1      % decrement counter

      jump sum        % iterate loop
```

TALx86

```
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax
```

sum's code type:

```
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
```

```
sum:
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }
     beq ebx, ecx    % if ebx=0, jump to ecx
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }
     add eax, ebx    % eax := eax + ebx

     sub ebx, 1      % decrement counter

     jump sum        % iterate loop
```

TALx86

```
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax
```

sum's code type:

```
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
```

```
sum:
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }
     beq ebx, ecx    % if ebx=0, jump to ecx
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }
     add eax, ebx    % eax := eax + ebx
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }
     sub ebx, 1      % decrement counter
    { eax: int32,   ebx: int32,  ecx : {eax: int32} }
     jump sum        % iterate loop
```

Modelling Calling Conventions

```
% sum: eax + 1 + 2 + … + ebx
%
% eax: accumulator
% ebx: counter
% ecx: continue with result in eax
```

sum's code type:

```
{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32}
}
```

a different calling convention:

```
∀α.{
  eax: int32,
  ebx: int32,
  ecx: {eax: int32, edx: α},
  edx: α
}
```

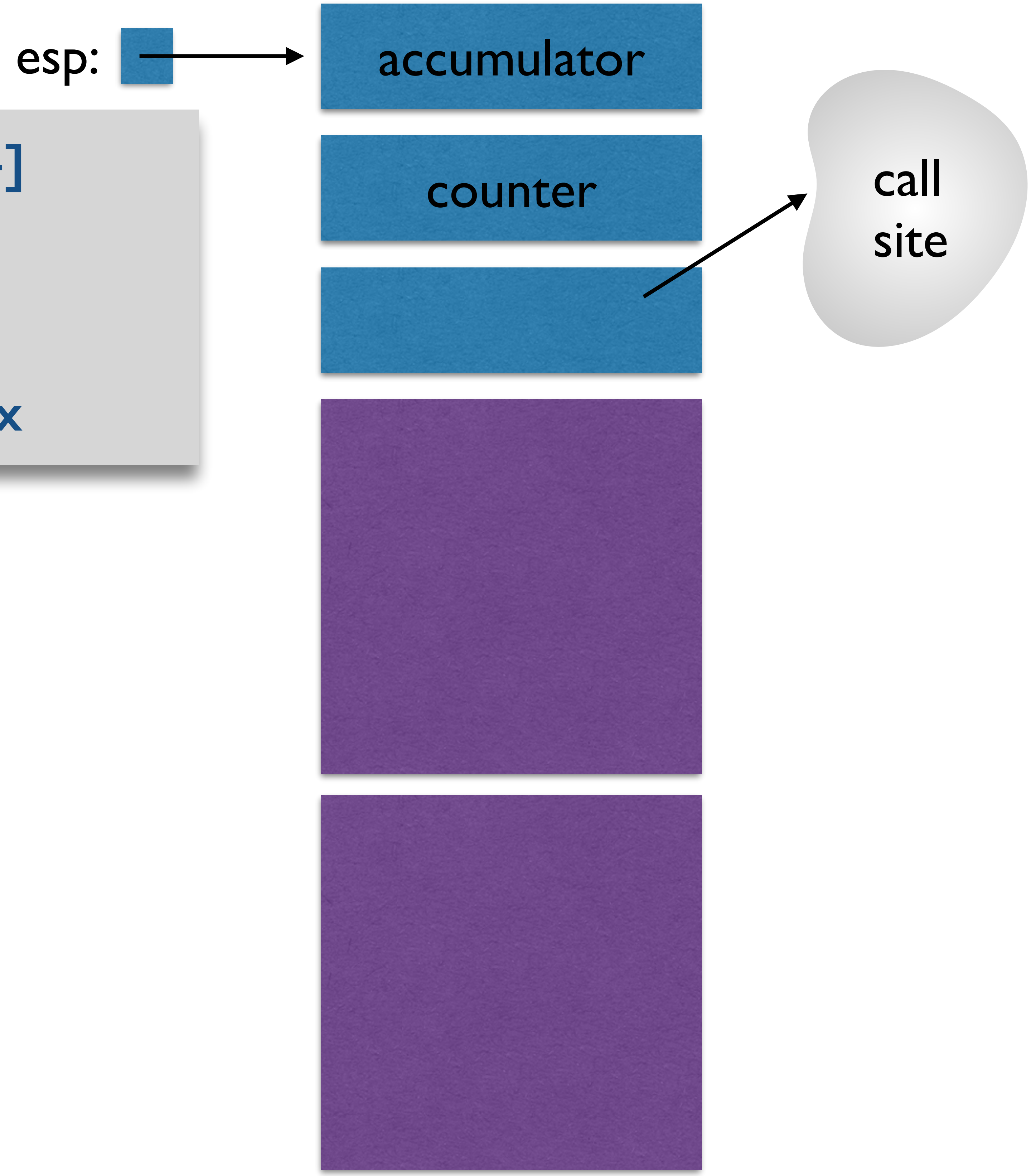
Callee (sum) saves register:

Type abstraction requires the callee to act parametrically in α

A Simple Proof Technique for Certain Parametricity Results. Karl Crary. ICFP 1999.

Extensions: Stack Typing

% sum: esp[0] + 1 + 2 + … + esp[4]
%
% esp[0]: accumulator
% esp[4]: counter
% esp[8]: continue with result in eax



Extensions: Stack Typing

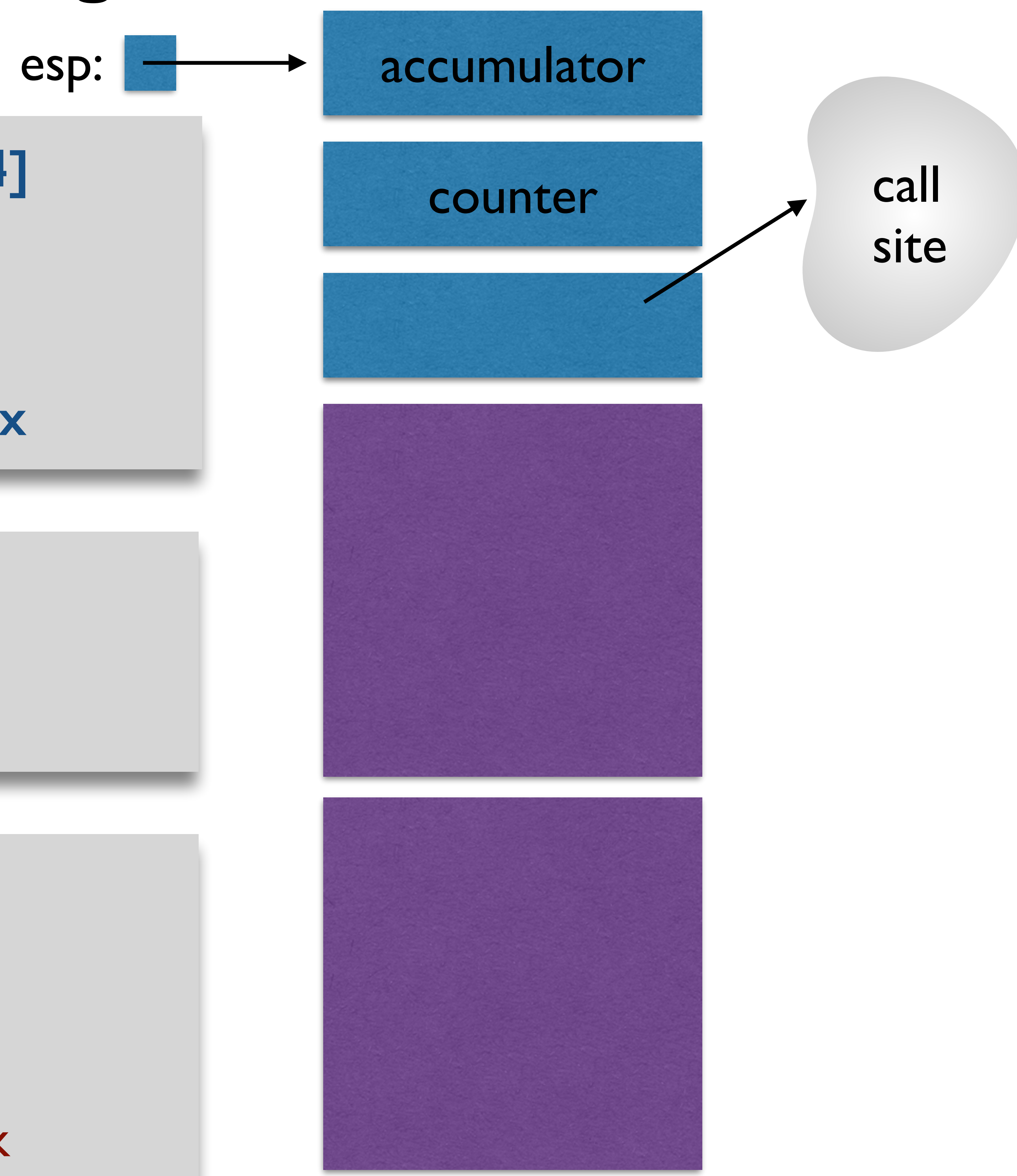
% sum: esp[0] + 1 + 2 + … + esp[4]
%
% esp[0]: accumulator
% esp[4]: counter
% esp[8]: continue with result in eax

modelling stacks s as lists:

s ::= nil | v :: s

stack types σ via an algebra of lists:

σ ::= nil	% empty stack
| τ :: σ	% a value on top
| ρ	% an abstract stack
| σ @ σ	% two stack segments

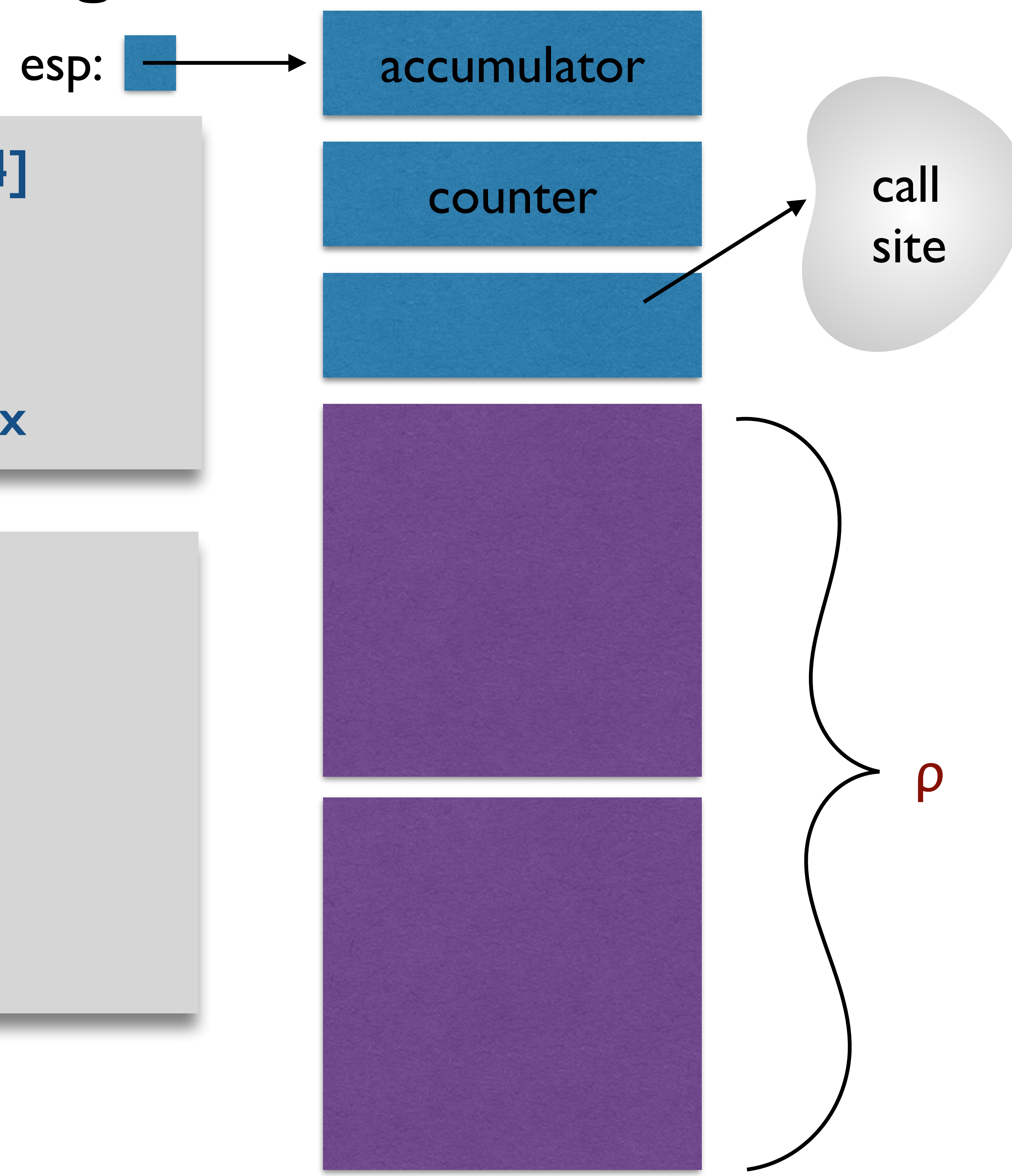


Extensions: Stack Typing

% sum: **esp[0]** + 1 + 2 + … + **esp[4]**
%
% **esp[0]**: accumulator
% **esp[4]**: counter
% **esp[8]**: continue with result in **eax**

stack-based sum's code type:

$\forall\rho.\{$
  esp: int32 ::
    int32 ::
    {**eax**: int32, esp: ρ} ::
    ρ,
$\}$

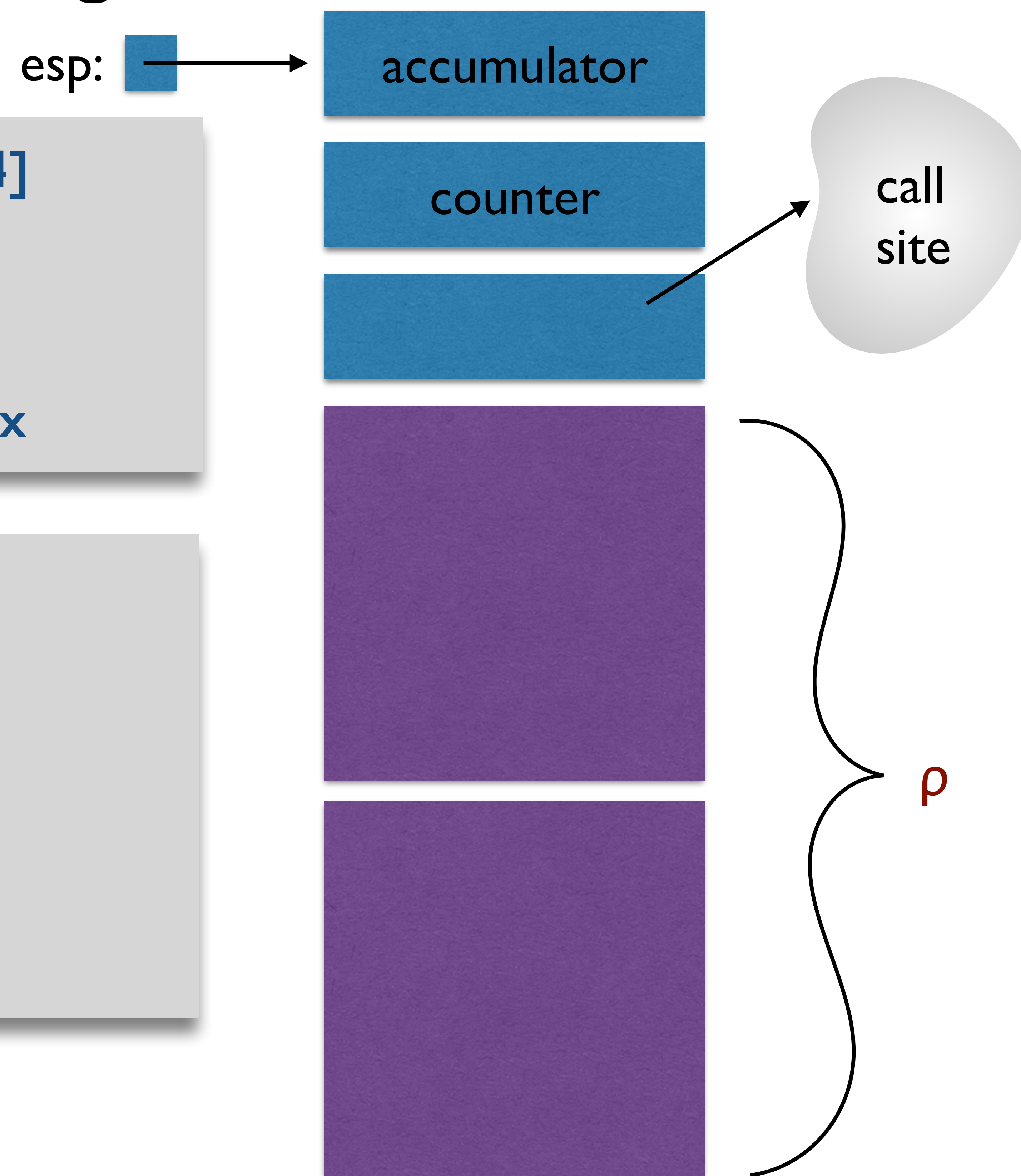


Extensions: Stack Typing

% sum: **esp[0]** + 1 + 2 + … + **esp[4]**
%
% **esp[0]**: accumulator
% **esp[4]**: counter
% **esp[8]**: continue with result in **eax**

stack-based sum's code type:

∀**ρ**.{
  esp: int32 ::
        int32 ::
        {**eax**: int32, esp: ρ} ::
        ρ,
}



Parametric polymorphism prevents
the callee from trampling on the caller's stack

A Simple Proof Technique for Certain Parametricity Results. Karl Crary. ICFP 1999.

TALx86 Summary

More types:

- For closures, data types, arrays, exceptions
- Types and kinds for describing object sizes, memory allocation and initialization
- Linking
- …

Moral of the story:

- Basic research in types reused in an extreme new setting
  - Impossible without syntactic proof techniques
- Biggest contribution:
  - Showing fully automatic proof of strong safety properties in general-purpose assembly is possible

Non-technical Take-aways

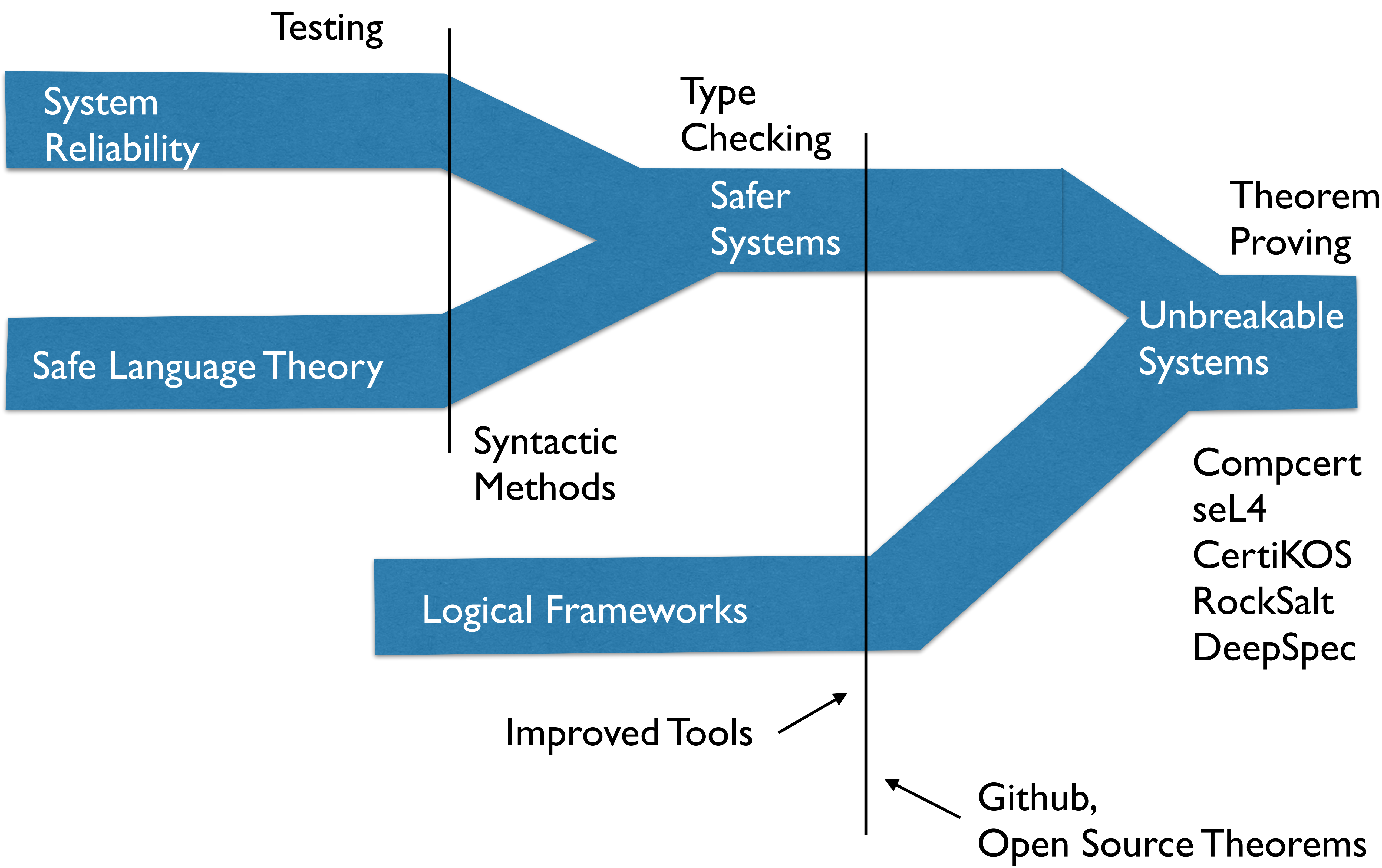
Learn a small number of highly reuseable skills really well.

- I learned one non-trivial proof technique:
  - Progress and Preservation
  - I practiced it over and over
- I learned how to develop small models:
  - idealized operational models with abstract objects
    - stacks, heaps, registers, …
  - tiny type systems, simple algebras
  - simplicity takes practice and experience
    - nobody ever uses or remembers the complicated stuff

I have used almost nothing else for the rest of my career.

(perhaps I'm lazy)

The Confluence Continues

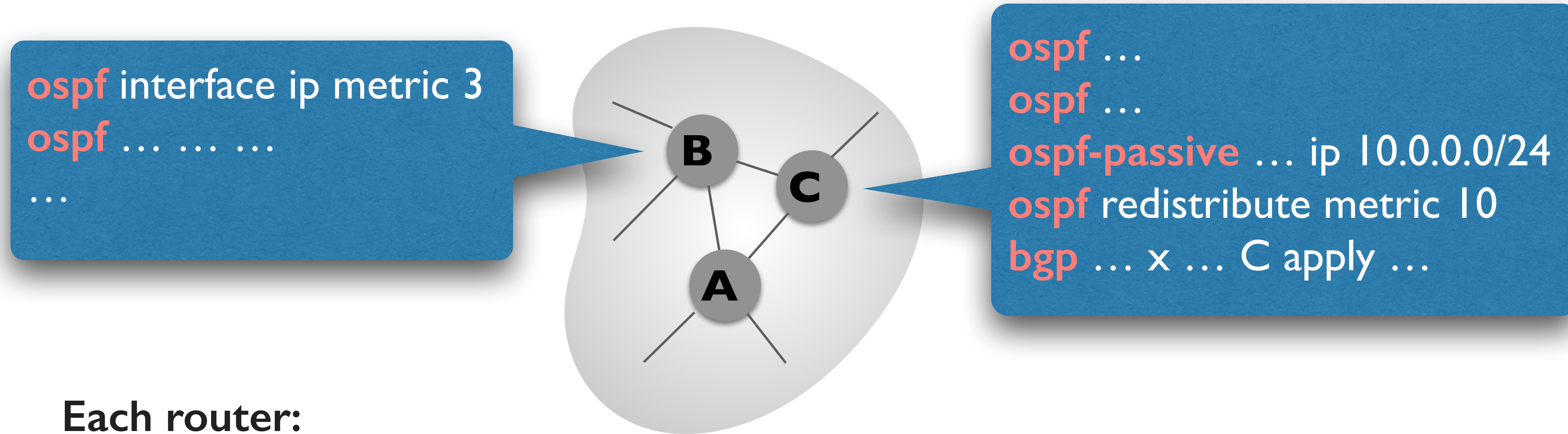


Professor Life: Making Friends
Confluences in Network Configuration

With Carolyn Jane Anderson, Ryan Beckett, Nate Foster, Michael Greenberg,
Arjun Guha, Stephen Gutz, Rob Harrison, Jean-Baptiste Jeannin,
Naga Praveen Katta, Dexter Kozen, Mathew Meola, Chris Monsanto,
Josh Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger,
Alec Story, Todd Warszawski

Network Configuration

Traditional Networks



Each router:

- maintains its own view of the world
- uses a standard protocol to communicate with neighbours and select routes

Network operators select from these standard, pre-defined protocols

- Operators supply parameters to configure them

Hardware vendors (eg, CISCO) control the software

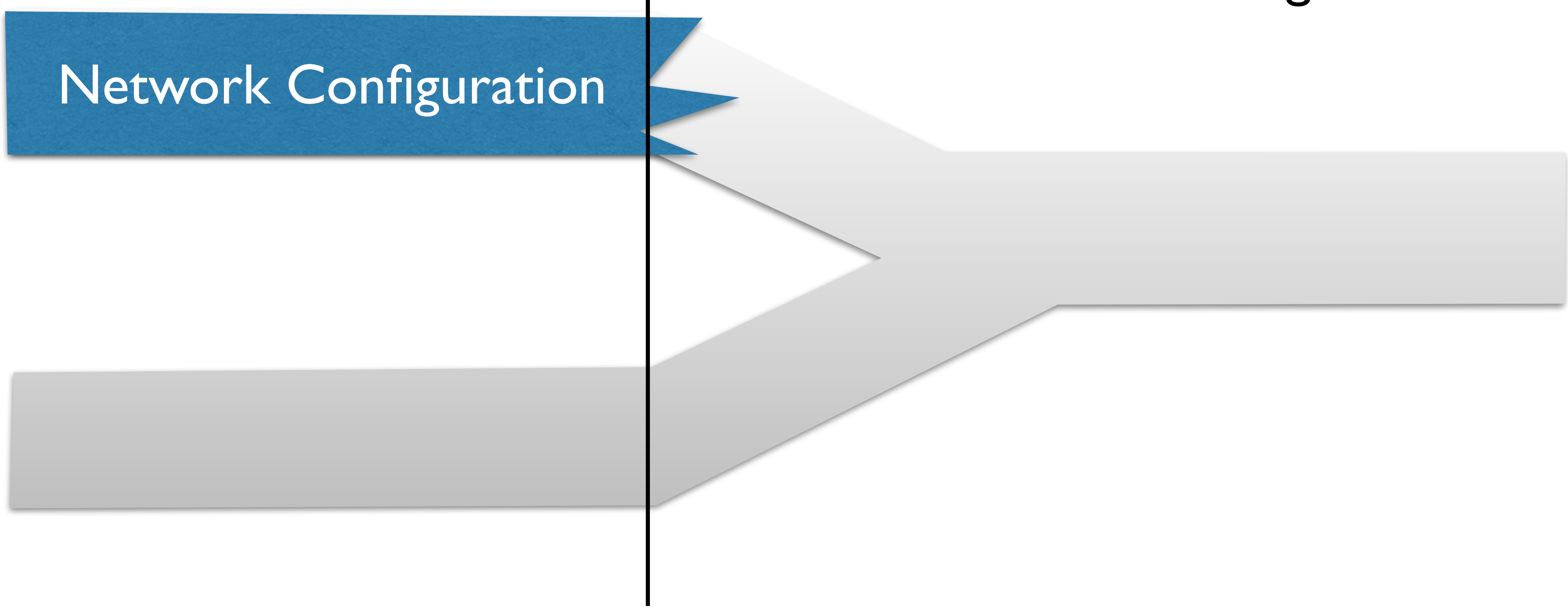
- Protocol standards evolve slowly

The Inflection Point

Technological & Economic Changes:

Data center infrastructure scales up

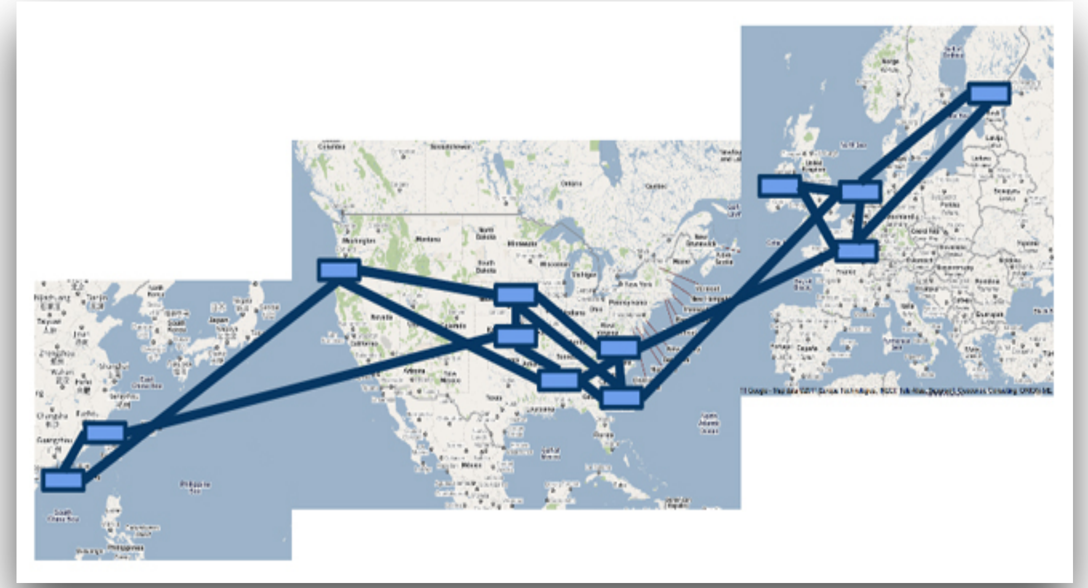
Owners of this infrastructure stand to gain from *customized* and *centralized* network control algorithms.



Connecting Inter-continental Cloud Services

Traditional WANs:

- No control over end hosts
- All bits treated the same
- 30-40% utilization achieved
  - overprovisioning for fault tolerance

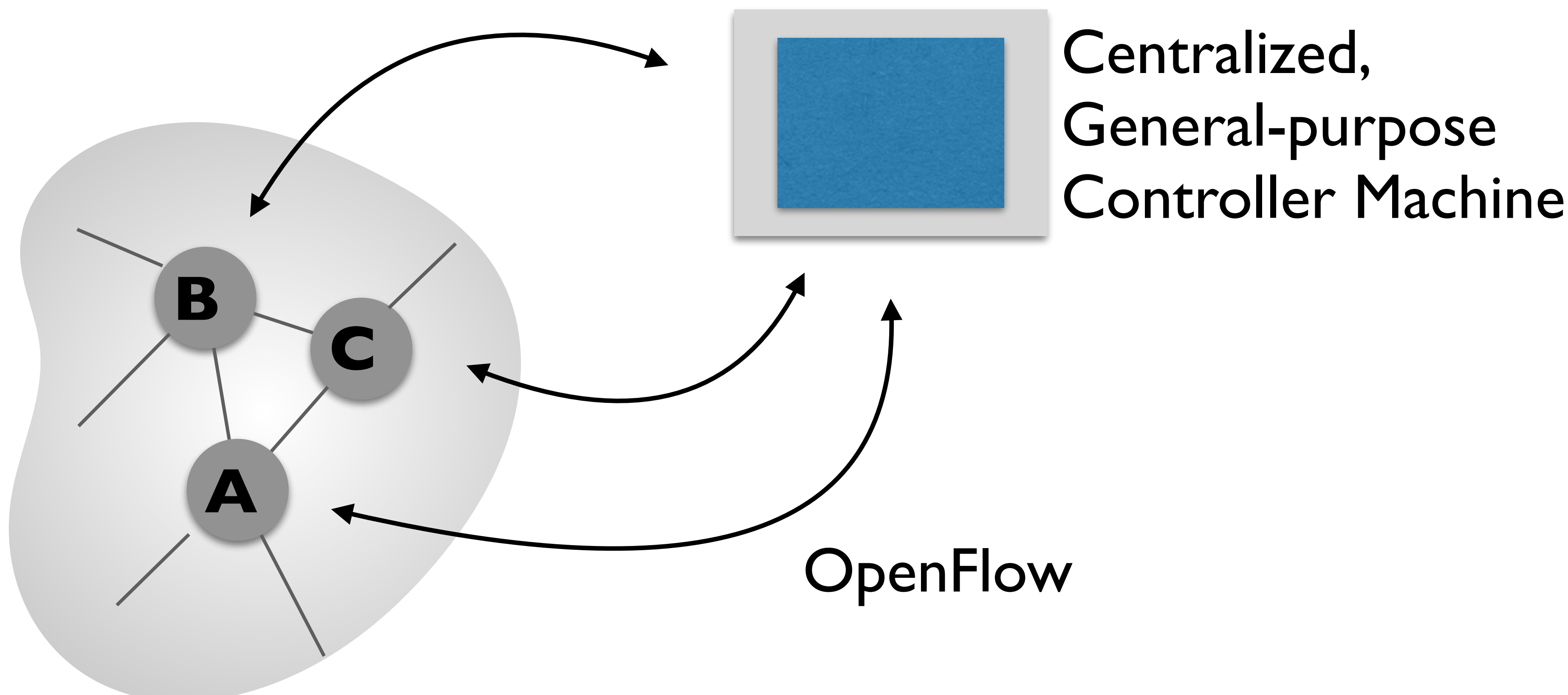


B4 WAN Connects Google's Data Centers:

- Control over end applications — limit their sending rate
- Multiple traffic classes, treated differently
  - user traffic: low volume, latency sensitive
  - big data synchronization: high volume, latency insensitive, fault tolerant
- Through centralized route control and traffic engineering, link utilization nears 100% on some links. Averages 70% or more throughout. 2x-3x cost savings.

B4: Experience with a Globally-Deployed Software-Defined WAN. Jain et al SIGCOMM 2013

Software-Defined Networking (SDN): The Technology Behind B4



Centralized controller plans routes using global information

- Rather than configuring distributed algorithms, the controller tells each switch how to forward, modify or drop packets directly
- OpenFlow: The new "network assembly language"
  - simple
  - yet expressive, capable of constructing any path

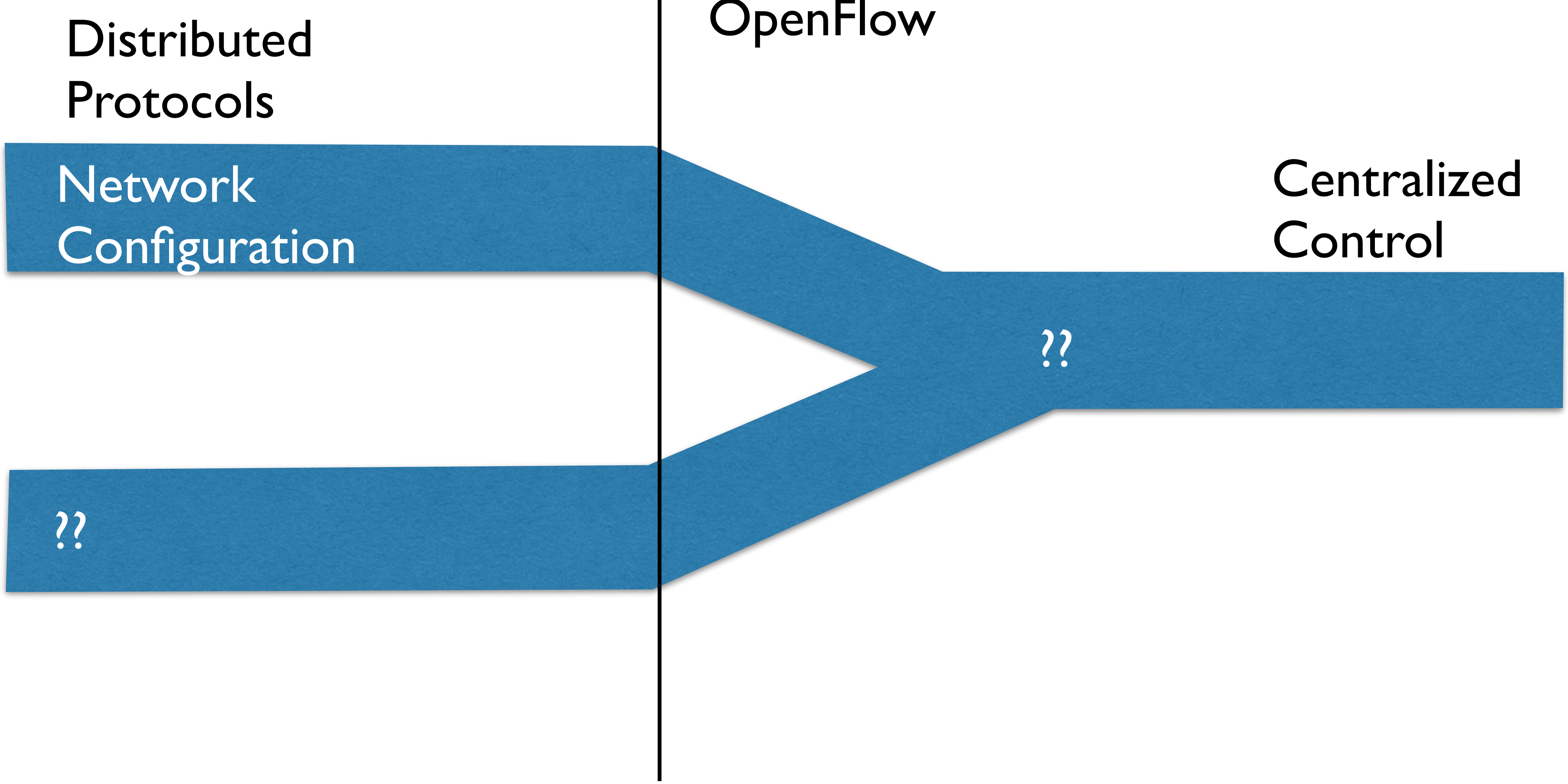
Confluences in Network Configuration

Economic Changes:

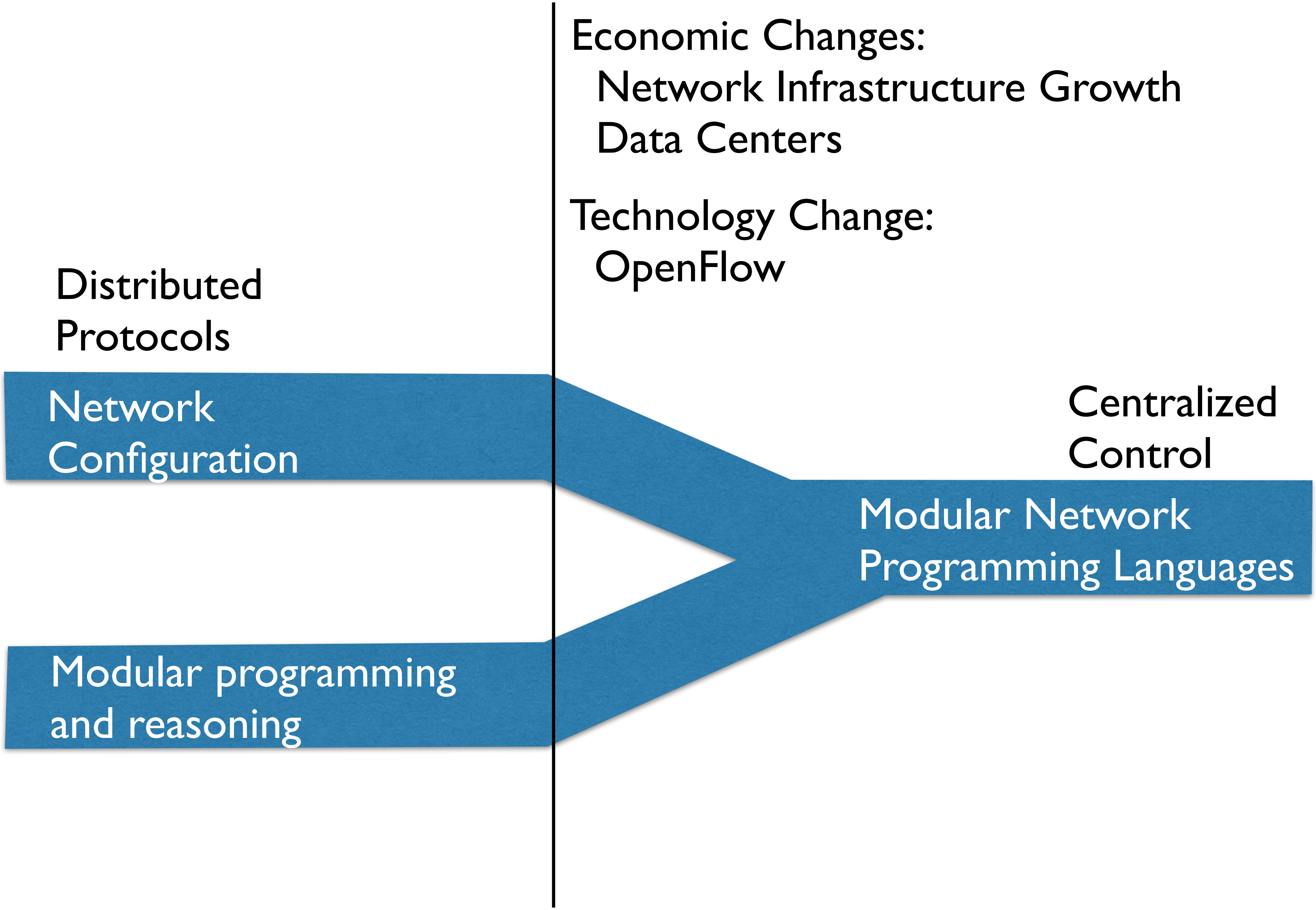
- Network Infrastructure Growth
- Data Centers

Technology Change:

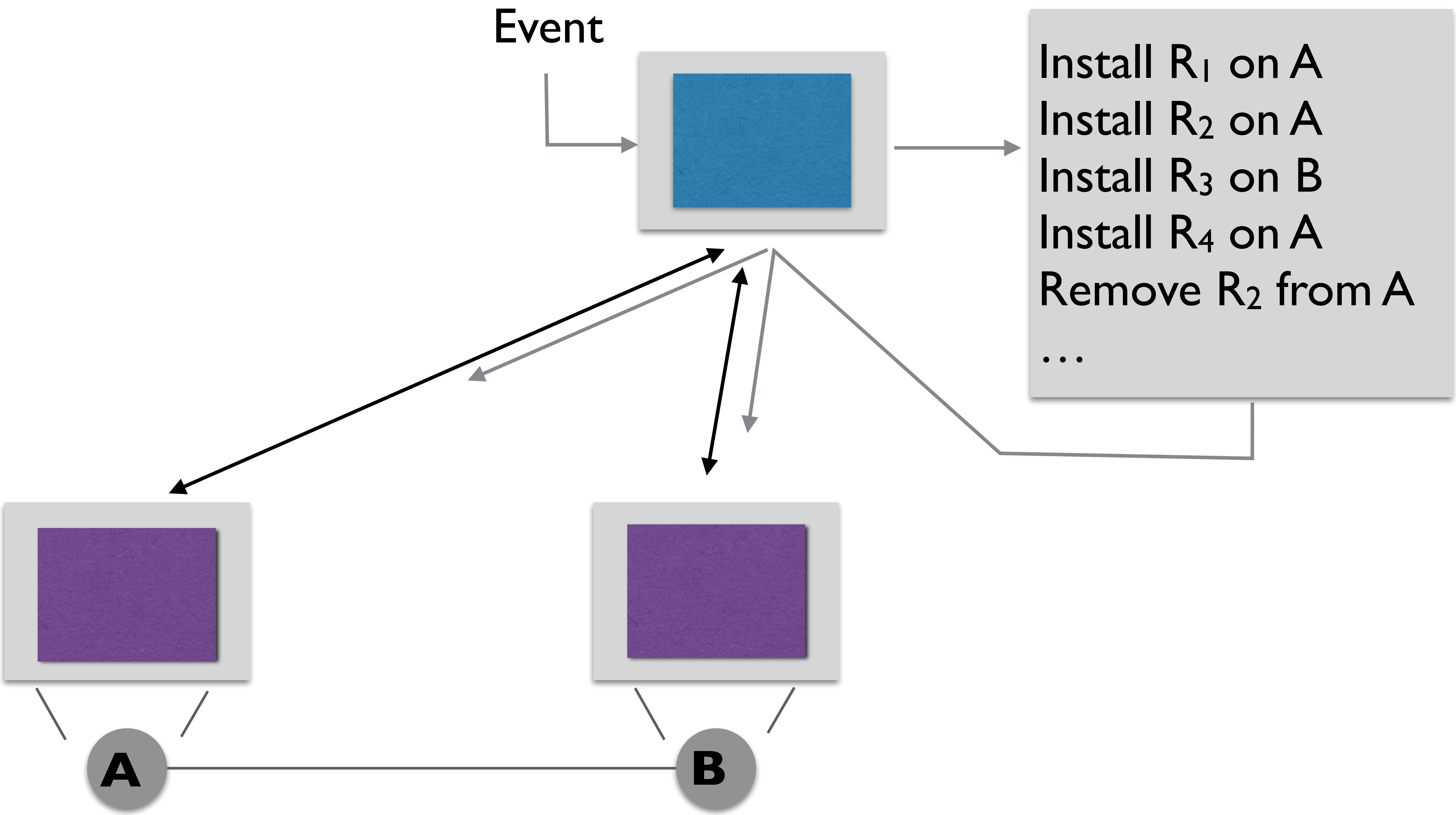
- OpenFlow



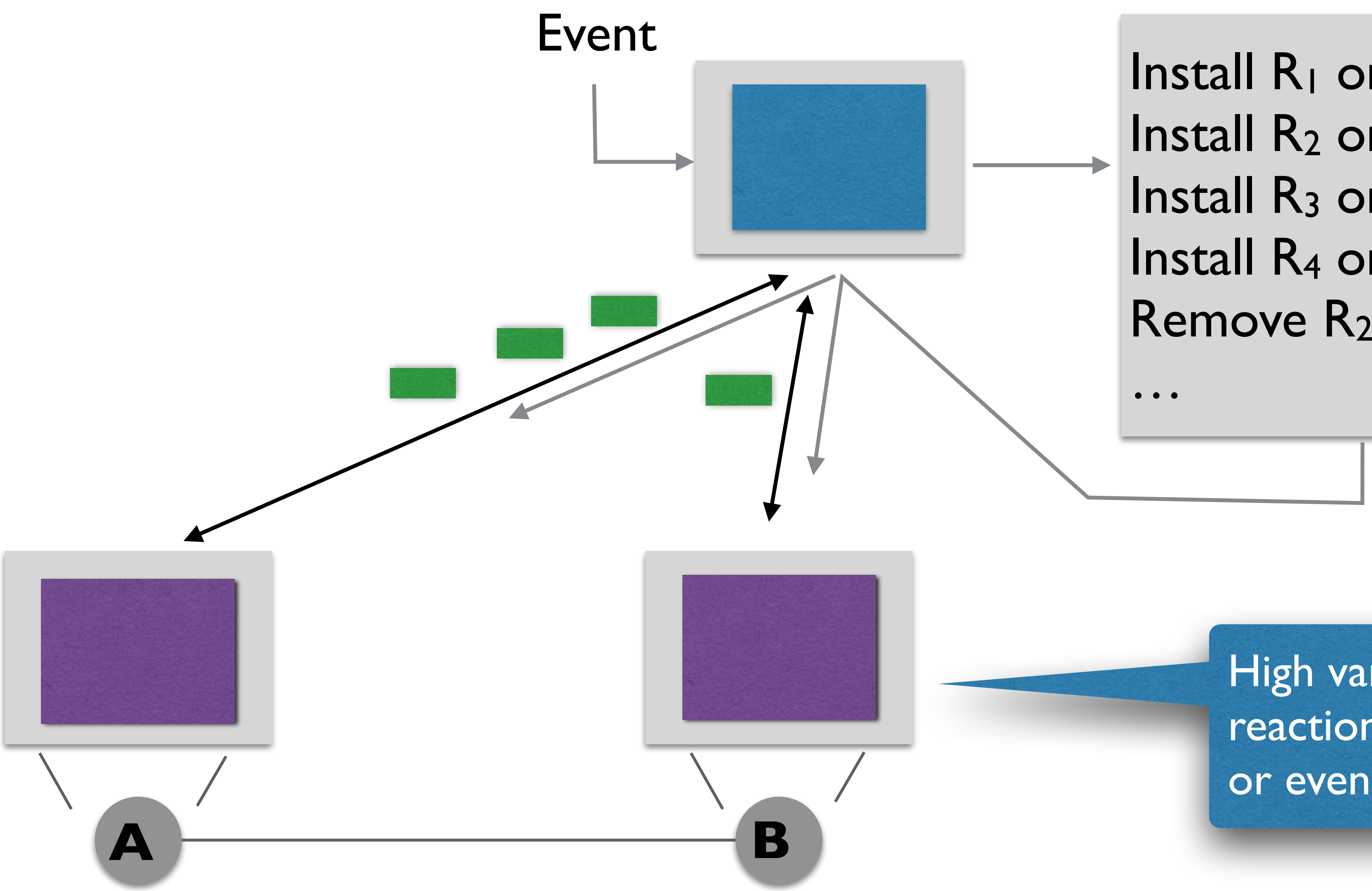
Confluences in Network Configuration



Software-Defined Network (SDN) Programming



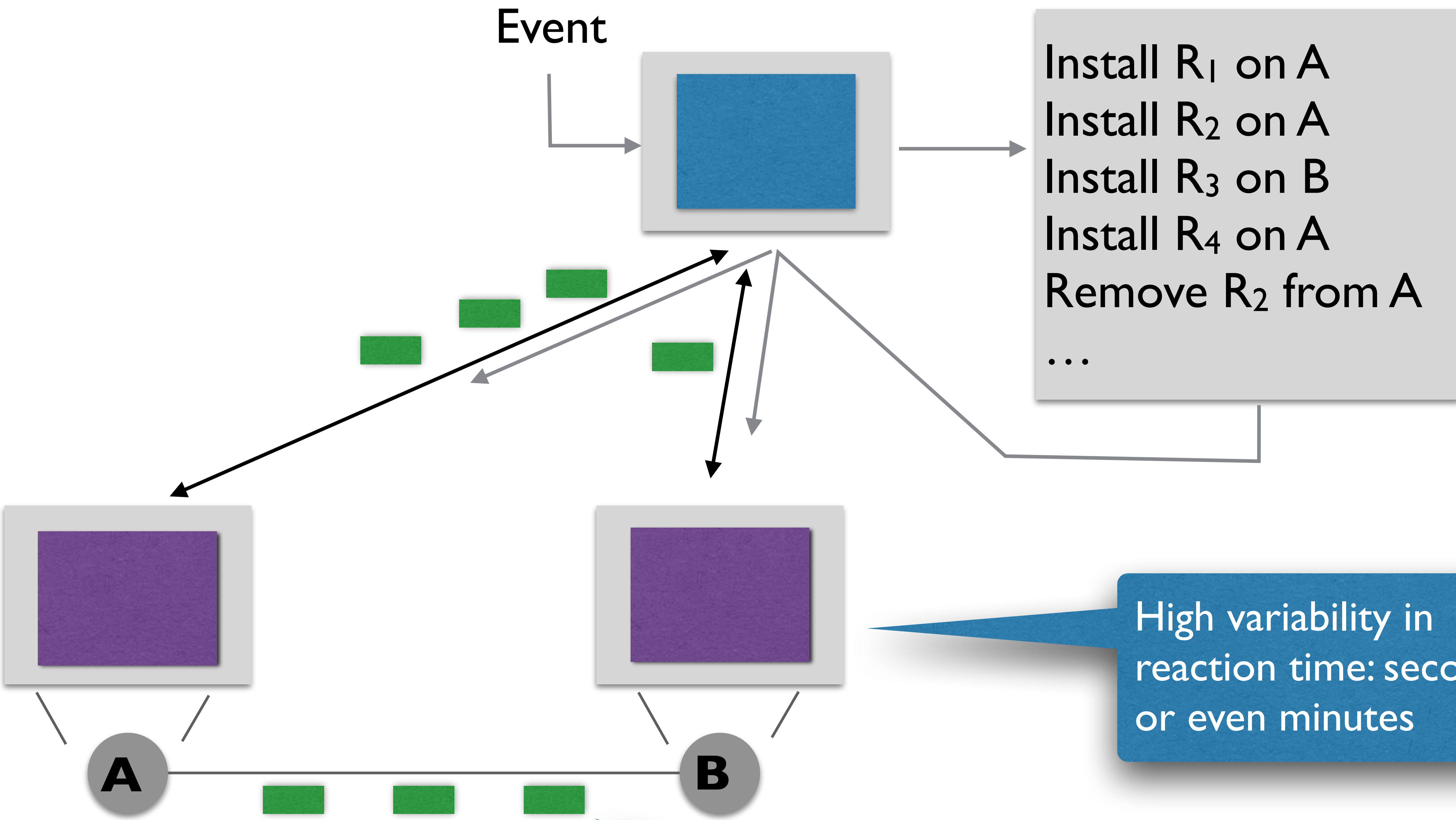
SDN Programming



Install R_1 on A
Install R_2 on A
Install R_3 on B
Install R_4 on A
Remove R_2 from A
…

High variability in reaction time: seconds or even minutes

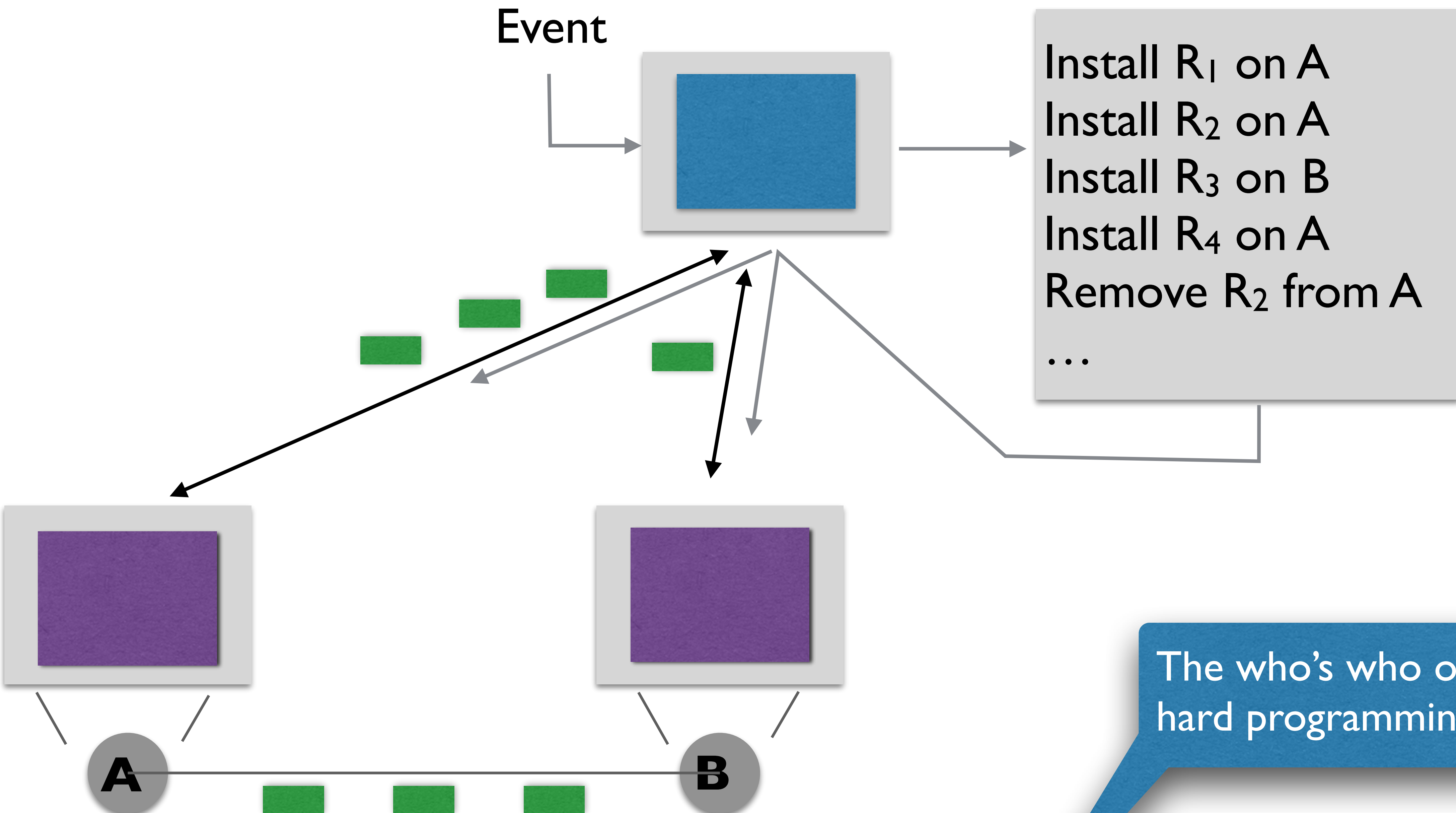
SDN Programming



High variability in reaction time: seconds or even minutes

At the same time, switch continues processing incoming packets at line rate

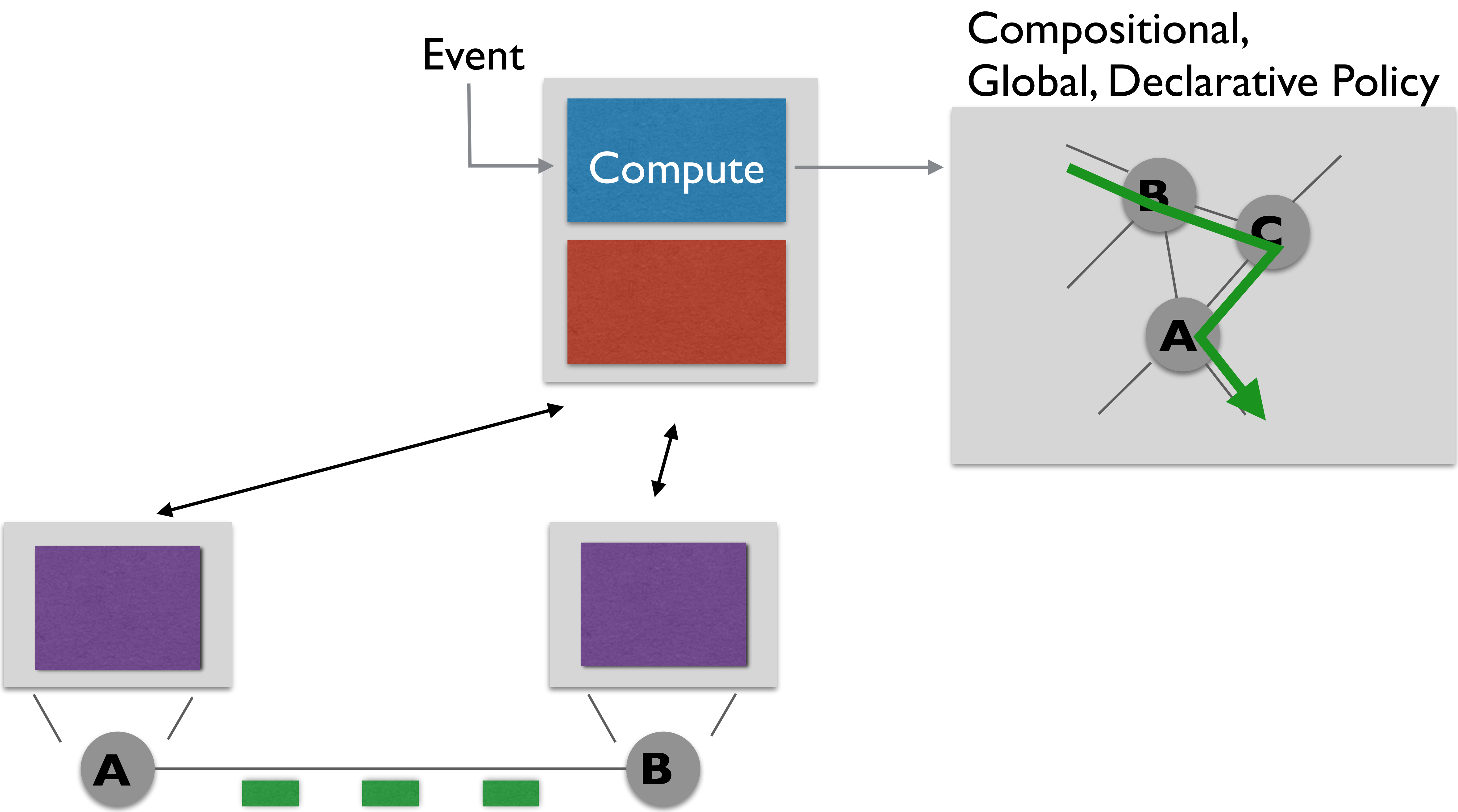
SDN Programming



The who's who of hard programming tasks

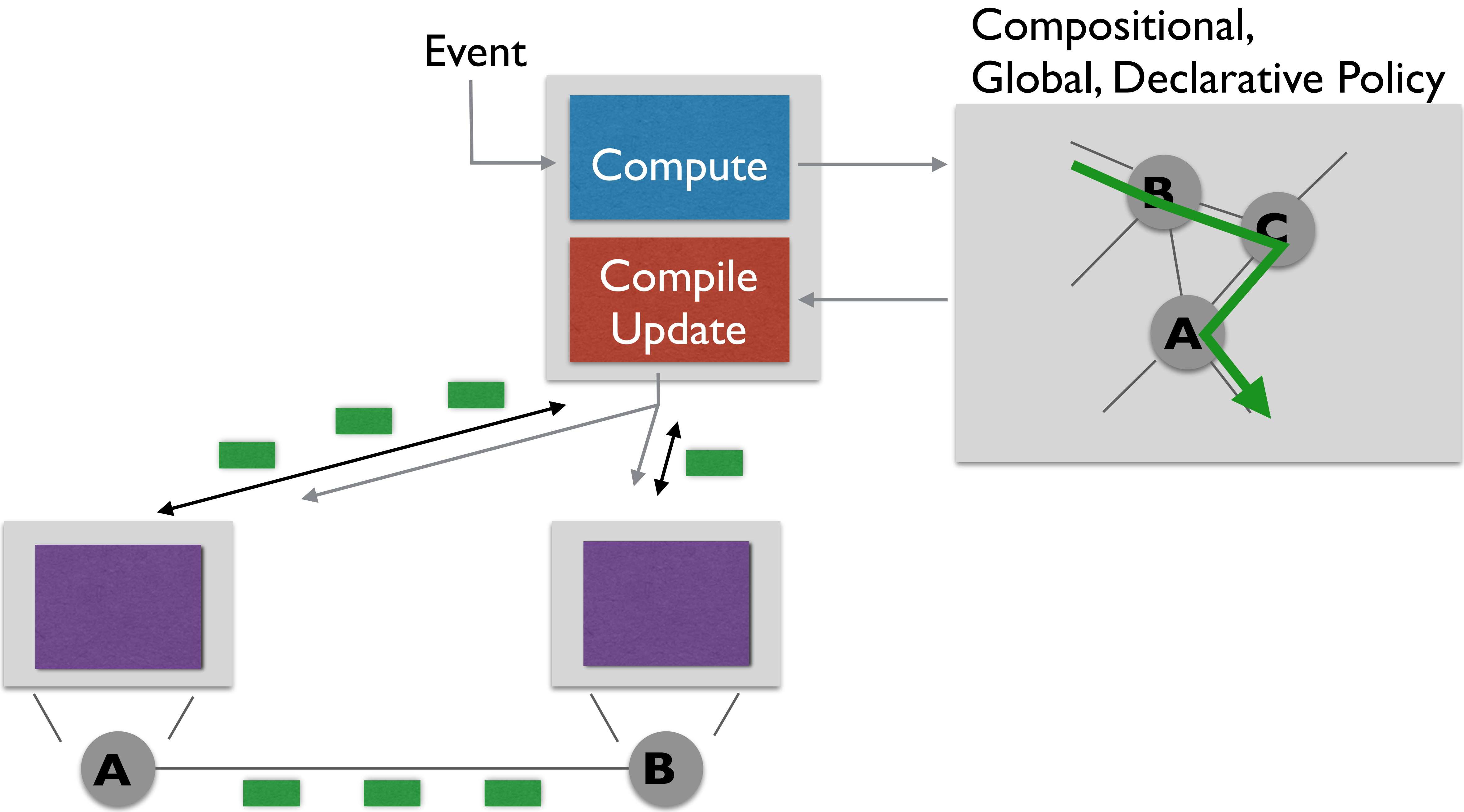
Early SDN: Event-driven, imperative, concurrent programming with distributed, stateful tables read asynchronously by other agents

Frenetic: Structured SDN Programming



Frenetic: A Network Programming Language. Foster, Harrison, Freedman, Monsanto, Rexford, Story, Walker. ICFP 2011

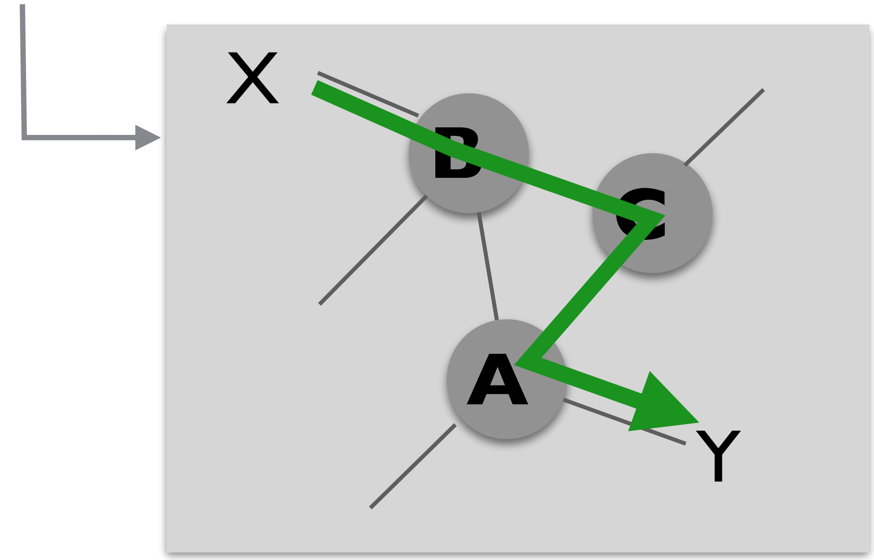
Frenetic: Structured SDN Programming



Frenetic: A Network Programming Language. Foster, Harrison, Freedman, Monsanto, Rexford, Story, Walker. ICFP 2011

Programmer's View

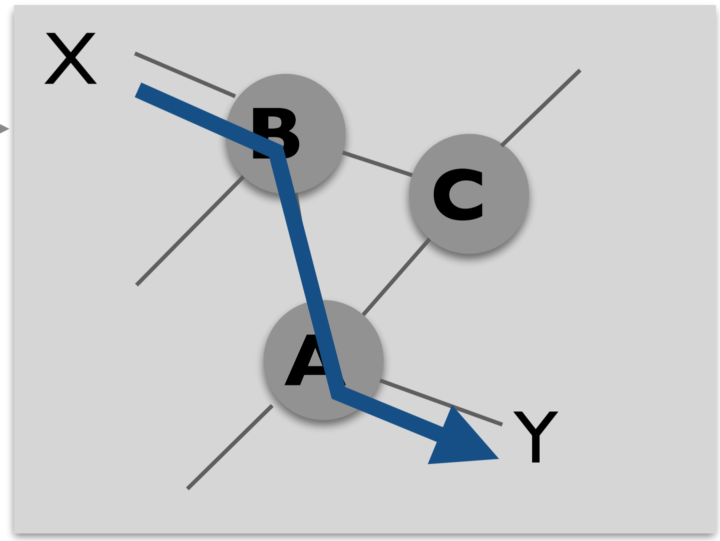
Event 1



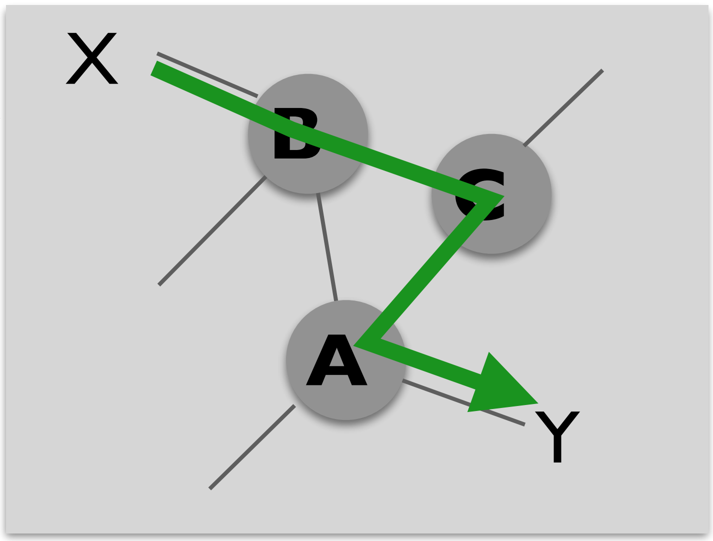
Programmer's View

…

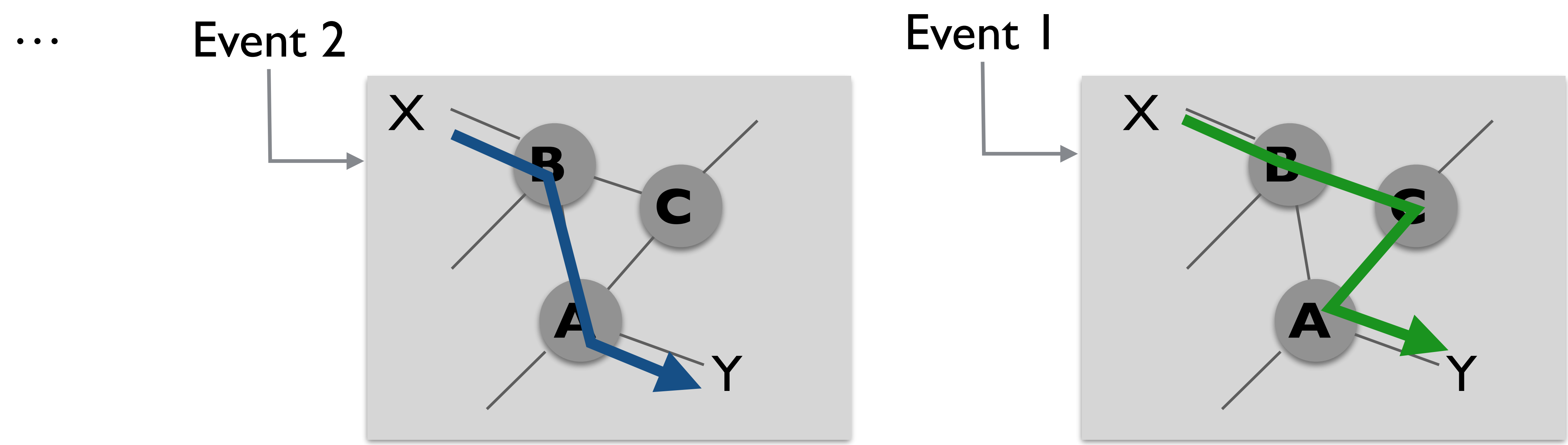
Event 2



Event 1



Programmer's View

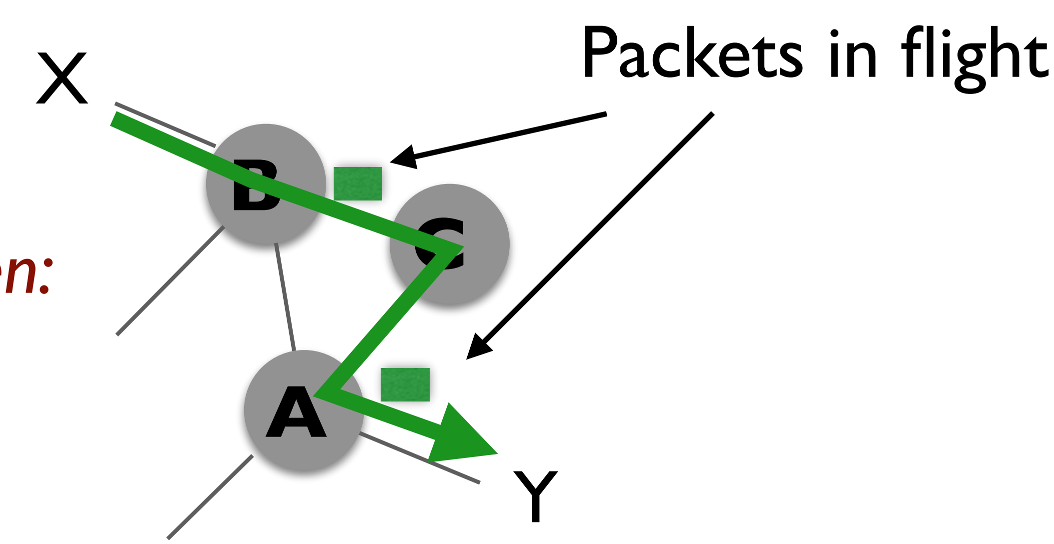


Underlying Physical Network

We need some protocol for updating switches.

If we aren't careful a lot of bad stuff could happen:

- packets from X to Y could be dropped
- packets could be mis-directed
- congestion?



Clearly, the protocol should preserve some "good" properties across updates

Preserving Properties

What kinds of properties?

- *Per-packet Path Properties (PPP)*: Any property of a single packet, its path through the network, and modifications along the way
  - Access control
  - Reachability
  - Way-pointing
  - But not congestion (a property of many packets)

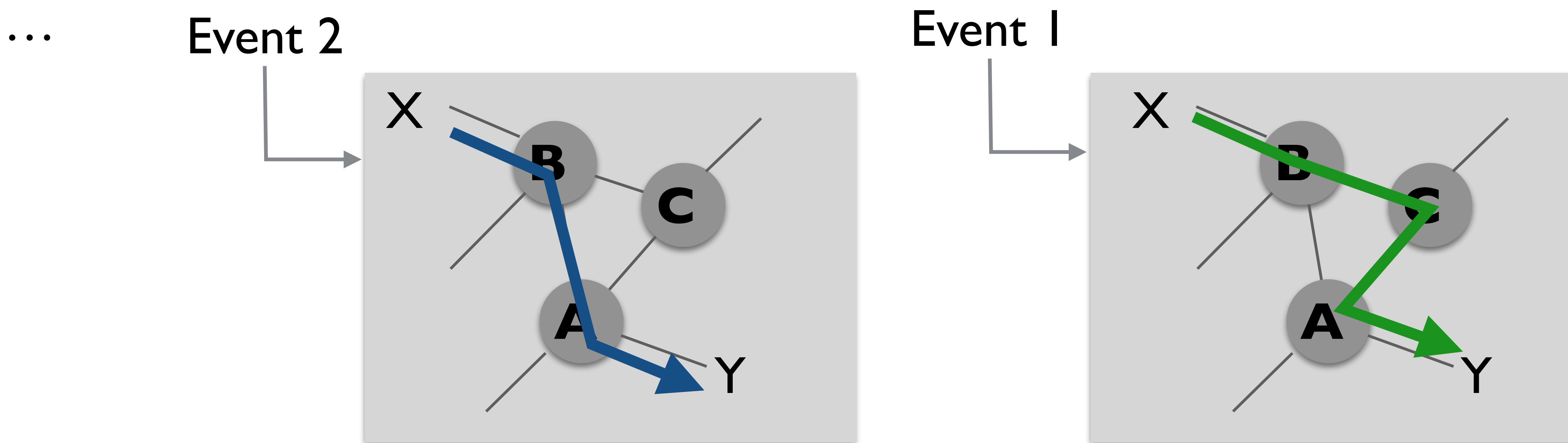
Which ones?

- *All of them*: Preserve any PPP shared by 2 consecutive policies
- *Advantage:* Programmers don't need to supply invariants
- *Advantage:* To check Inv is preserved forever, check all policies independently

How?

- *Per-packet Consistent Update:* Ensure every packet traverses either the old policy or the new policy, not some mixture of both

Implementation Mechanism: 2-phase Commit



Preprocess every policy:

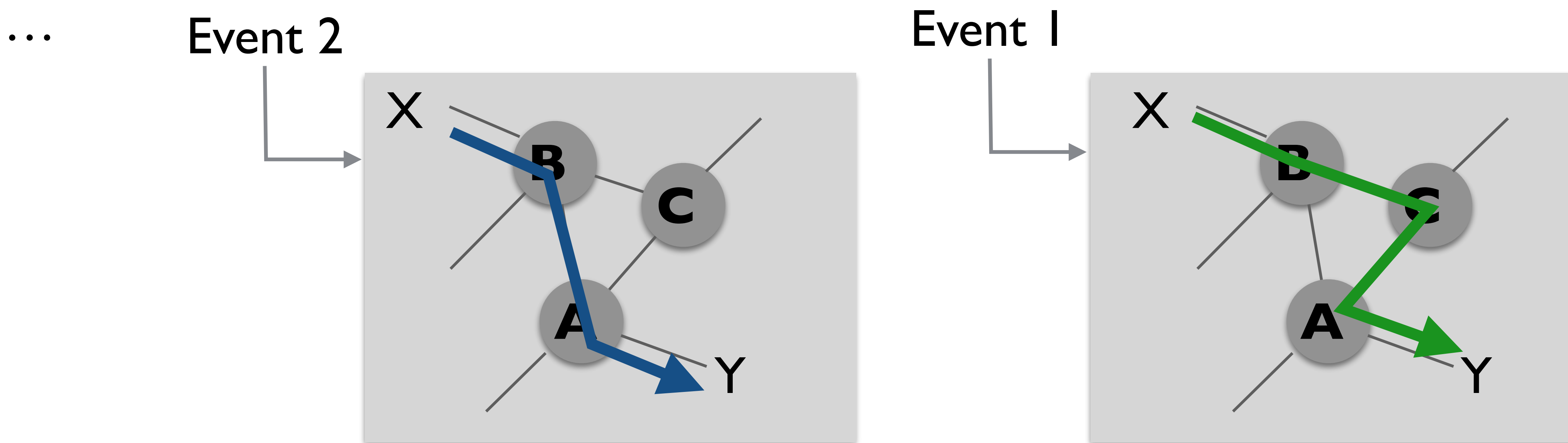
- Entry locations stamp policy version number on packets (green/blue)
- Internal location apply their policy if the packet carries the right number

To update from green to blue:

- *Phase 1:* Add new blue rules to internal switches, while packets continue to be stamped green and are processed by green rules
- *Phase 2:* Overwrite entry location green-stamping rules with blue-stamping rules

Abstractions for Network Update. Reitblatt, Foster, Schlesinger, Rexford, Walker, SIGCOMM 2012.

Improvements and Refinements



Incremental updates trade time for space [Katta et al., HotSDN 2013]

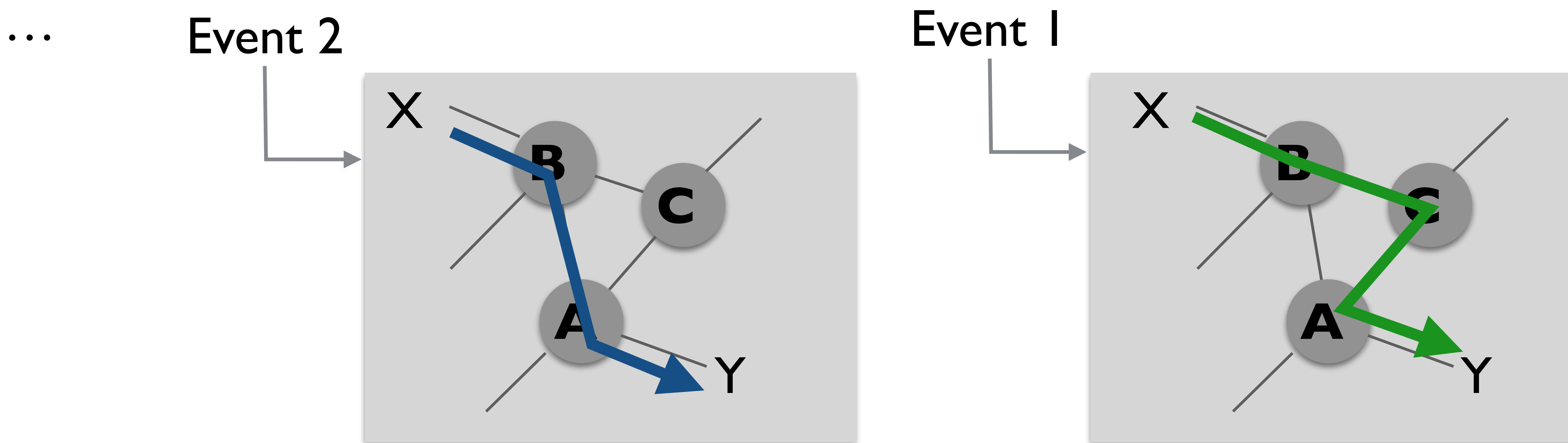
Updates with congestion control [Hong et al., SIGCOMM 2013]

Dynamic update scheduling improves update time [Jin et al., SIGCOMM 2014]

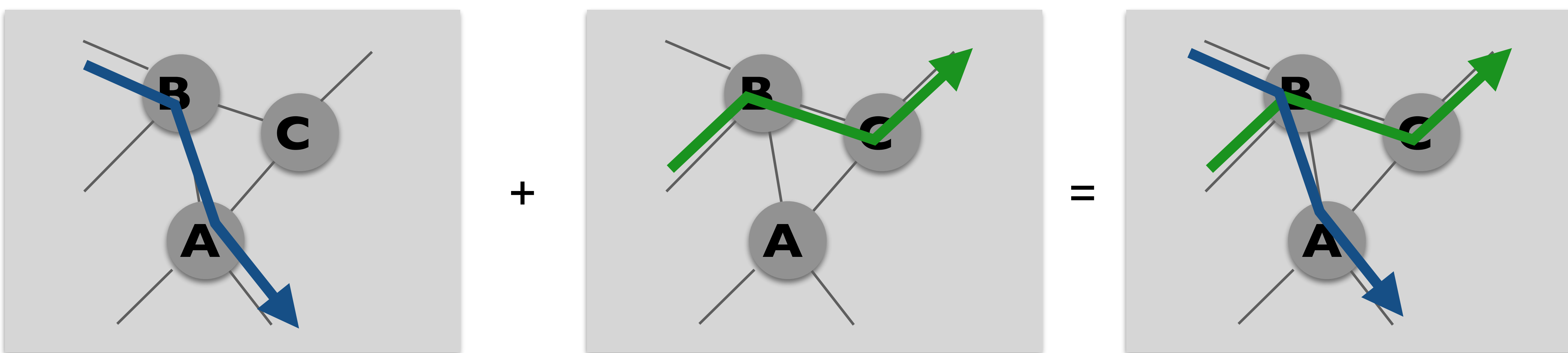
Preserving user-supplied invariants instead of all invariants improves update time and space! [McClurg, PLDI 2015]

…

Consistent Updates: Modular Reasoning in Time



Frenetic Policy Lang's: Modular Reasoning in Space



Frenetic [ICFP 11], NetCORE [POPL 12], Pyretic [NSDI 13], NetKAT [POPL 14],
SDX [SIGCOMM 14], Fast NetKAT [ICFP 15], Concurrent NetCORE [ICFP 15],
CoVisor [NSDI 15], Kinetic [NSDI 15], Probabilistic NetKAT {ESOP 16], Path Queries [NSDI 16] …

Technical Take-aways

The networking community has embraced language-based approaches to network configuration.

ACM Symposium on SDN Research (SOSR) sponsored by SIGCOMM topics include:

- Programming languages, verification techniques and testing techniques for SDN

P4: A Language-based "OpenFlow 2.0"

- start: a PL/networking group [SIGCOMM CCR 2014]
- now: 33 member organizations (as of Dec 14, 2015)
- several PL folks providing feedback

MOOC: Software-Defined Networking

- Nick Feamster (Georgia Tech → Princeton)
- 870 students doing assignments, survey
- 217 full-time network operators
- 79% preferred Kinetic [NSDI 15] to current approaches
- 84% agreed it helped make it easier to verify policies

Non-Technical Take-aways

Sometimes research is all about the detailed result:

- Progress and Preservation

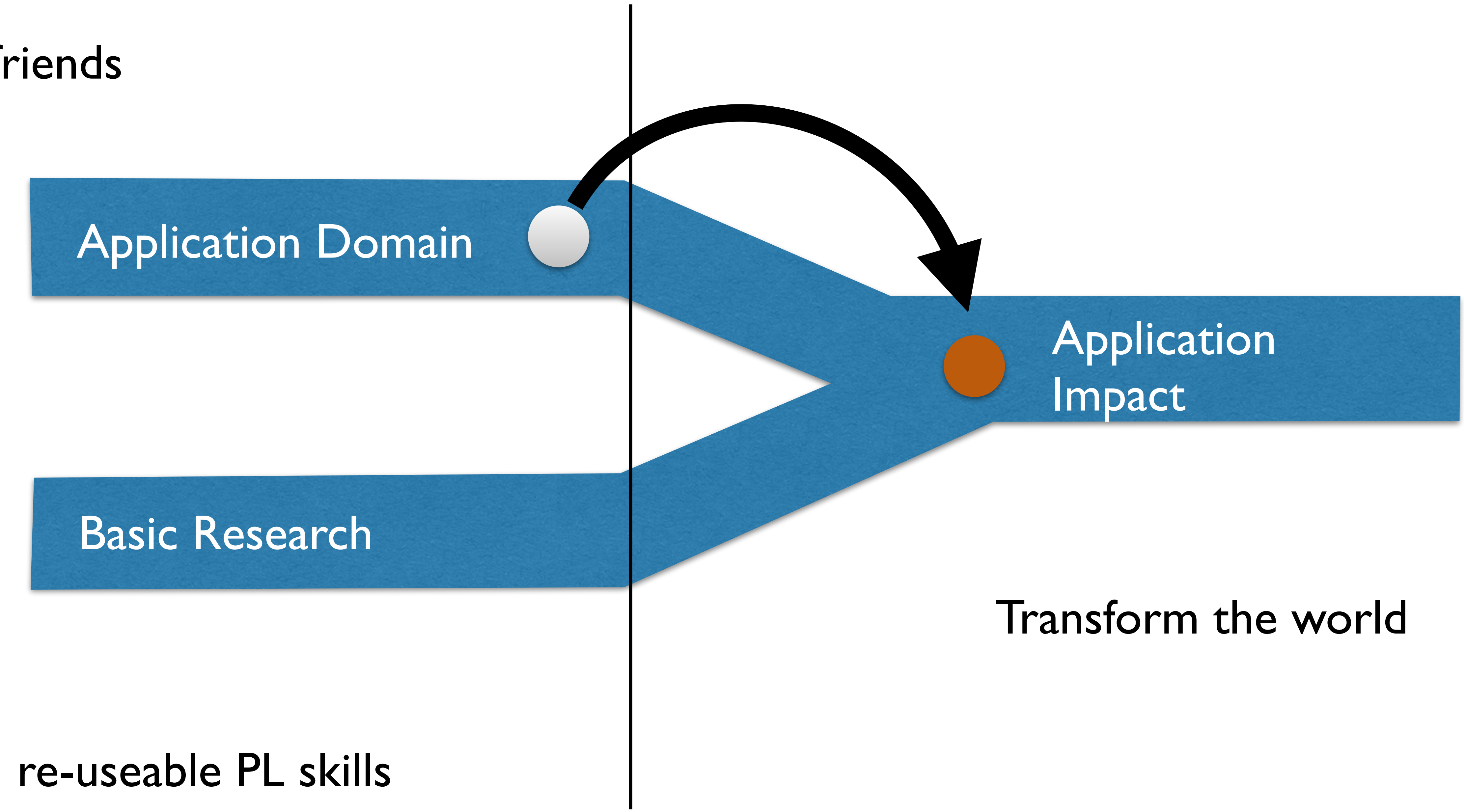
But sometimes it is people and communication that matter most:

- We got in to SDN early when no one had written any programs. How?
  - Our colleague Jen Rexford was at the forefront of the area
    - She developed the intellectual precursors to SDN at AT&T
    - She spotted the SDN inflection point
    - She was open-minded
  - We wrote a grant together
  - We had beyond-brilliant colleagues (Nate Foster and others)
- Then we got mind share. How?
  - Jen gave early an keynote talk at the Open Networking Summit
  - Followed up the next year by Nate Foster
  - Jen gave many, many industrial talks; she has many friends

Moral: Make Friends

Summary:
Confluences in Programming Language Research

Make friends



Learn re-useable PL skills

Be open-minded
Watch for the inflection points


thank you