# Broom: Safe Region-based Memory Management for BigData Processing Systems

with G Ramalingam, Kapil Vaswani, Dimitrios Vytiniotis, Jana Gicheva, Michael Isard, Steve Hand & Ionel Gog

Microsoft

In this talk ...

can be framed and solved

How Memory Safety as Type Safety in Broom environment

# Memory Safety as Type Safety in Broom

# Broom

# Broom

Optimized software stack for big data pipelines

DryadLinq Naiad CScale

data processing pipelines for big data

compute count on
(select .. from ...)

- large
- semi-structured
- distributed

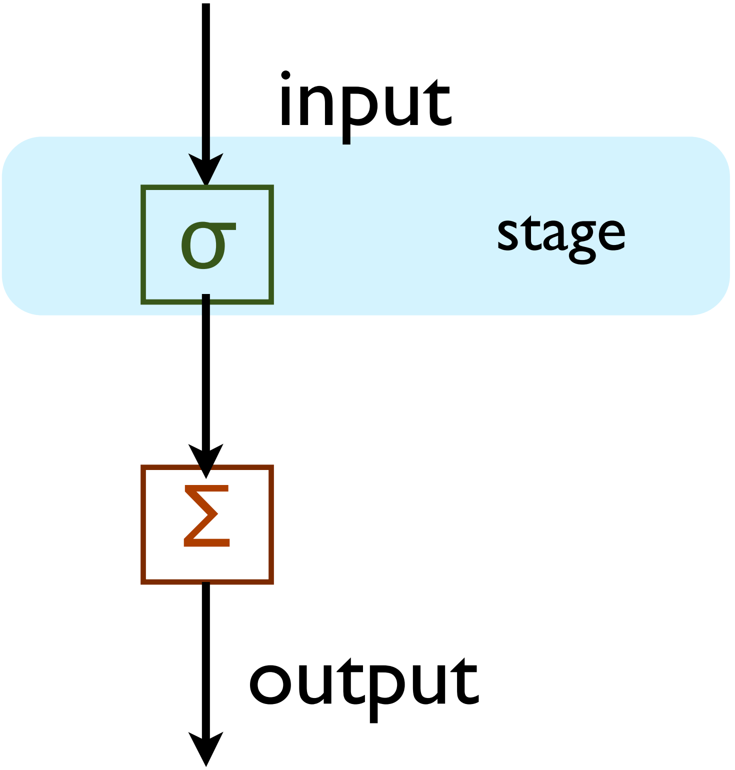
# Broom

```
compute count on
(select ... from ... )
```

# Broom

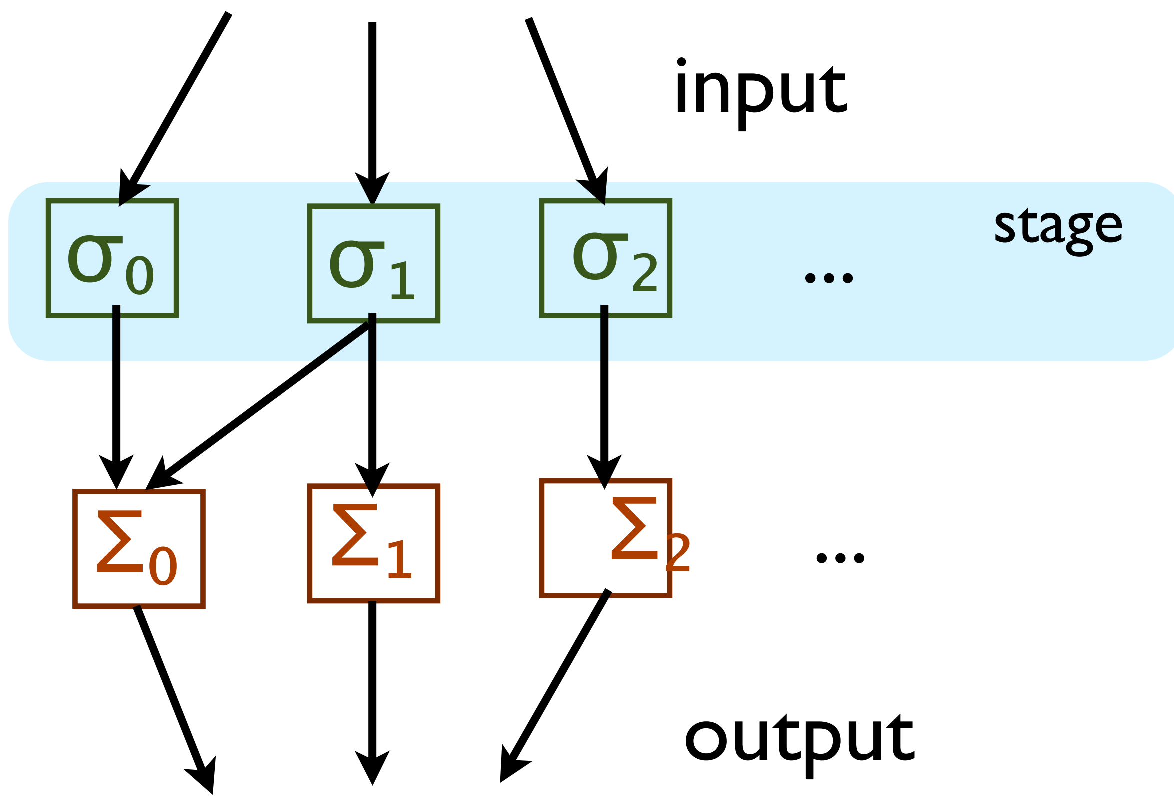
compute count on
(select ... from ...)

computations can be staged



# Broom

compute mean on
(select … from … )




computations can be staged

… where each stage is a data parallel computation

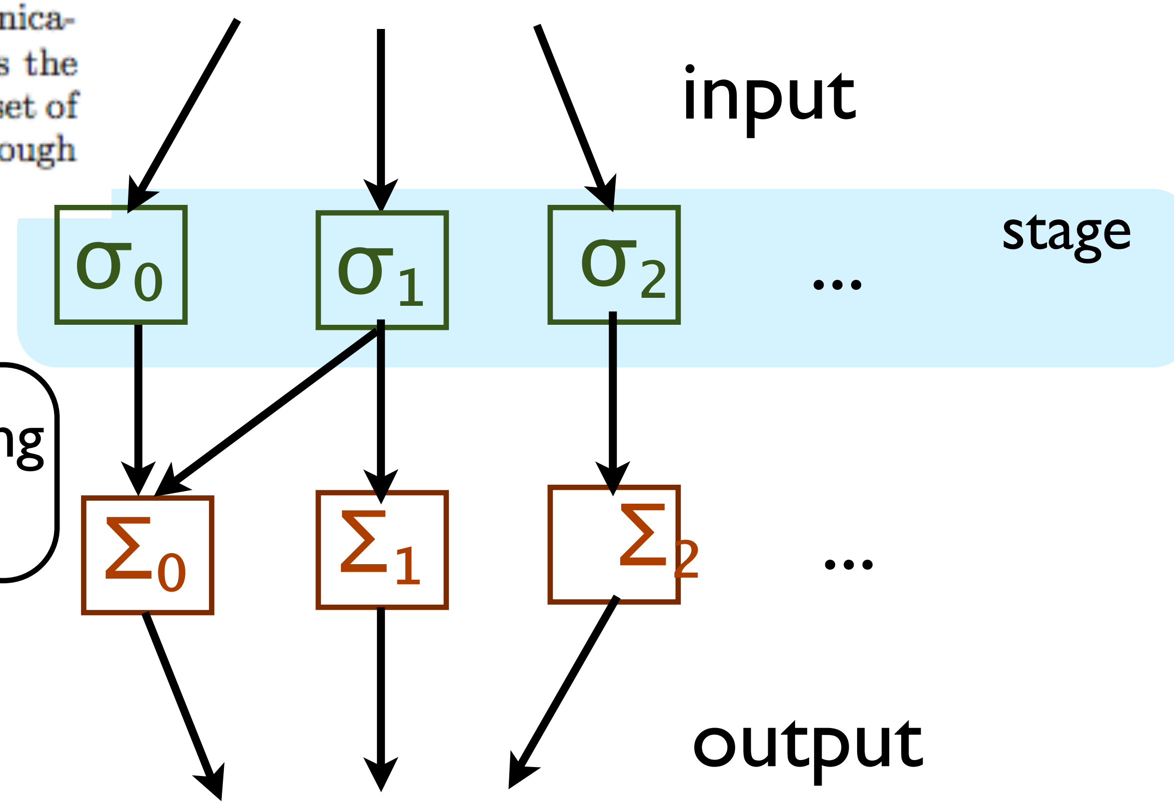
Most big data computations can be represented as dataflow DAGs

# Broom

Optimized software stack for $\underbrace{\text{big data pipelines}}$

DryadLinq Naiad CScale

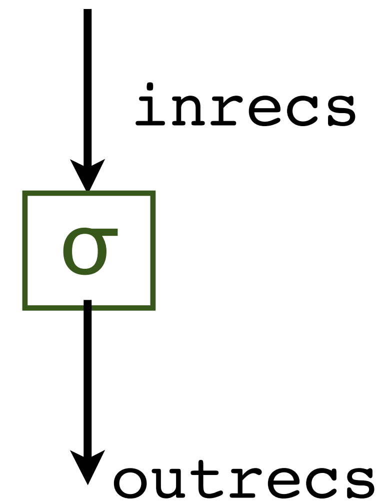
Dryad is a general-purpose distributed execution engine for coarse-grain data-parallel applications. A Dryad application combines computational “vertices” with communication “channels” to form a dataflow graph. Dryad runs the application by executing the vertices of this graph on a set of available computers, communicating as appropriate through files, TCP pipes, and shared-memory FIFOs.




computation vertices communicating via message-passing channels

Actors

# An Example Select Actor $\boxed{\sigma}$




select … from …

```
class selectActor {
  void select(Func<Record,OutRec> udf,
                              Batch<Record> inrecs) {
      var outrecs = new Batch<OutRec>();
      foreach var x in inrecs {
          var y = udf(x);
          outrecs.append(y);
      }
      transfer(outrecs, this.getDownStreamActor());
  }
}
```

# Broom

Current software stack for big data pipelines

General purpose:

★ Garbage collection

★ Scheduling

★ I/O management

Not necessarily optimized for distributed dataflow pipelines

# Broom

Optimized software stack for big data pipelines

More performant run-time by exploiting specific characteristics of dataflow programs

# Broom

for this talk ...

Memory $\underbrace{\text{Optimized}}_{}$ software stack for big data pipelines

More memory performant run-time by exploiting specific characteristics of dataflow programs

The aim is to bring down the GC overhead

# Broom Memory Management

“The basic concept of regions is very old, first appearing as early as 1967 in Douglas T. Ross's AED Free Storage Package” - wikipedia

Broom adopts a well-known memory management scheme called region-based memory management

“A region is a logical container for objects that obey some memory management discipline”

- Hicks, Morrisett et al., Cyclone, ISMM’04

For eg: The heap region, A stack frame

# Regions

We can create more specific regions, if we are aware of specific memory behaviour of programs

In Broom:

- ★ Data in memory that needs to be sent as a message to a downstream actor should'nt be deallocated unless the downstream actor is remote and the message is sent.
- ★ Once an actor finishes *acting* on a message:
  - ★ Incoming message can be deallocated
  - ★ Any objects created while in *action* can be reclaimed.

regions!

# Broom Regions

After observing the memory behaviour of dataflow programs, we introduced different kinds of regions in Broom:

- ★Transferable dynamic regions
- ★Non-transferable static regions:
  - ★Actor-scoped
  - ★Action-scoped
  - ★Block-scoped / Scratchpad

can be transferred by an actor to another actor

deallocated as soon as action finishes execution

# Static Broom Regions

- ★ Lifetime determined by lexical scope
- ★ Usually hold temporary objects

R0

R1

```
newAlloc {
    var x = new LinkedListNode<int>(2);
    newAlloc {
        var xs = new LinkedList<int>();
        xs.addAtHead(x);
    } xs gets deallocated here
} x gets deallocated here
```

# Transferable Broom Regions

- ★ Dynamic Lifetime
- ★ First class citizens - can be passed to, and returned from fns

```
var r = new Region<Batch<Record,OutRec>>;
openAlloc r {
   var msg = new Batch<Record,OutRec>()
   r.set(msg);
}
...
r.transfer(this.getDownStreamActor());
```

upstream actor

```
void onRecv(Region<Batch<Record,OutRec>> inr) {
    open inr {
       ...
    }
    inr.giveUp();
}
```

downstream actor

r

# An Example Select Actor $\boxed{\sigma}$

select … from …

```
class selectActor {
  void select(Func<Record,OutRec> udf,
                                Batch<Record> inrecs) {
      var outrecs = new Batch<OutRec>();
      foreach var x in inrecs {
          var y = udf(x);
          outrecs.append(y);
      }
      transfer(outrecs, this.getDownStreamActor());
  }
}
```

# Select using Broom Regions

```
void select(Region<Batch<Record>> rin,
                    Func<Record,OutRec> udf) {
    var rtrans = new Region<Batch<OutRecord>>();
    open (rin) {
       var batch = rin.GetData();
       openAlloc (rtrans) {
          rtrans.SetData(new Batch<OutRecord>());
       }
       newAlloc {
         foreach var x in batch {
            var y = udf(x);
            openAlloc (rtrans) {
               var z = rtrans.SuckObject(y);
               rtrans.GetData().append(z);
            }
         }
       }
   }
   rin.GiveUp();
   rtrans.Transfer(this.getDownStreamActor());
}
```

Memory regions have independent life times, and cross-region pointers can become dangling

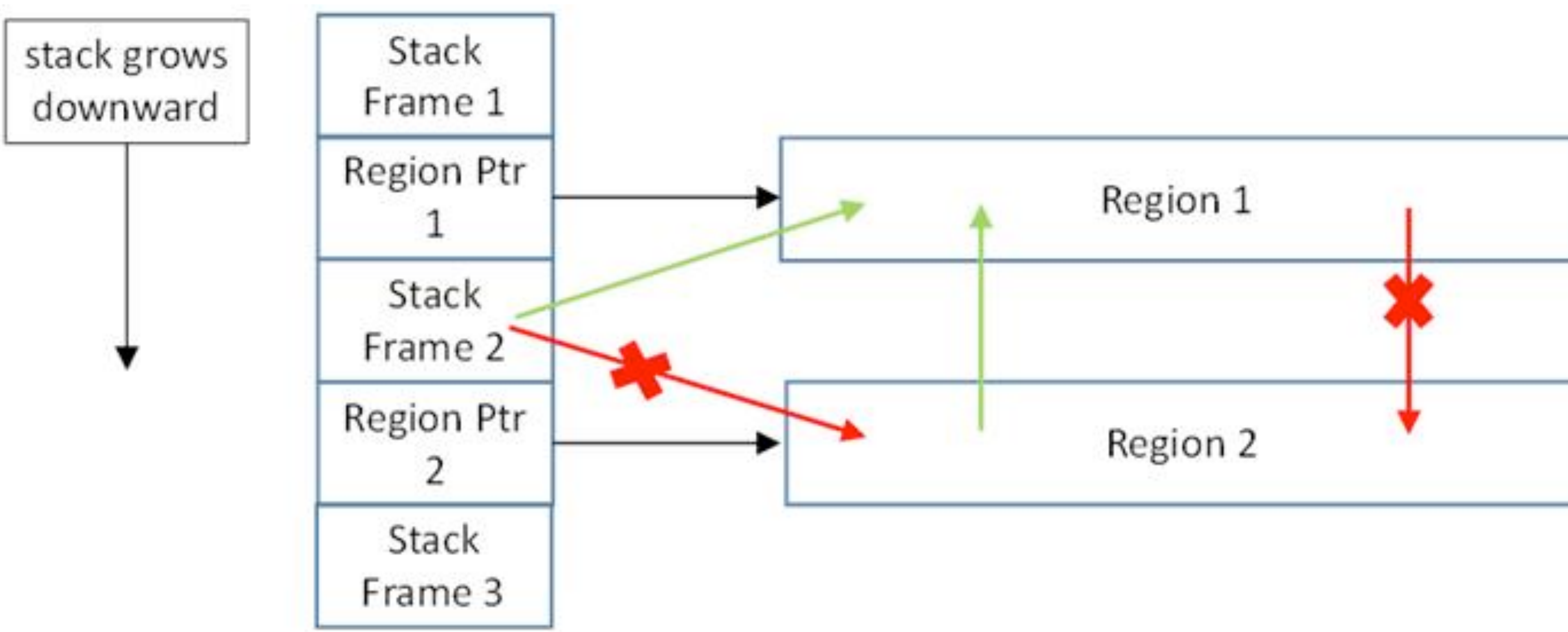
R1

R2

```
newAlloc {
    ...
    newAlloc {
        ...
    }
}
```

R1 “outlives” R2: `R1⩾R2`

Therefore, any references from R1 to R2 are unsafe



It is very easy to shoot oneself in foot.

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

```
var g;
newAlloc {
  var x = new A();
  g = new Func(y => x.m());
}
g();
```

It is easy to create dangling references

... especially with higher-order functions

```
void foo() {
  var rtrans = new Region<Batch<OutRecord>>();
  openAlloc rtrans {
    ...
  }
  return;
}
```

Space leaks are also possible.

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

References from live regions to dead regions

It is easy to create dangling references

most of the probable

Can we catch dangling references by analyzing the code at compile time?

Yes, through a region type system

Tofte, Talpin, Implementing a call-by-value λ-calculus using a stack of regions, POPL'94

# The Region Type System - Intuitions

R0 R1

```
newAlloc {
  var x = new A();    /* x : A<R0> */
  newAlloc {
    var y = new B(); /* y : B<R1> */
    x.f = y;
  }                     /* ∃(R≥R0). x.f:B<R> */
  print (x.f);
}
```

$$\underbrace{\texttt{x.f = y OK?}}_{\text{No!}}$$

# Region Type Checking - Modus Operandi

```
class B {} ⟶ class B<ρ0> { }
```

```
newAlloc<R0> {
    var x = new B<R0>();    /* x : B<R0> */
}
```

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

# Region Type Checking - Modus Operandi

```
class B {}
```

```
class B<ρ0> { }
```

```
class A {
  B f;
}
```

```
class A<ρ0> {
  B f;
}
```

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

# Region Type Checking - Modus Operandi

```
class B {}        →    class B<ρ0> { }

class A {         →    class A<ρ0,ρ1> {
  B f;                   B<ρ1> f;
}                      }
```

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

# Region Type Checking - Modus Operandi

```
class B {}
```

```
class B<ρ0> { }
```

```
class A {
  B f;
}
```

```
class A<ρ0,ρ1|ρ1≽ρ0> {
  B<ρ1> f;
}
```

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

# Region Type Checking - Modus Operandi

```
class B {}          →   class B<ρ0> { }

class A {           →   class A<ρ0,ρ1|ρ1≽ρ0> {
  B f;                    B<ρ1> f;
}                       }
```

```
newAlloc {                  → newAlloc<R0>
  var x = new A();          → new A<R0,R?>()
  newAlloc {                → newAlloc<R1>
    var y = new B();        → new B<R1>()
    x.f = y;
  }
  print (x.f);
}
```

but R1 doesn't exist here!

```
x.f : B<R?>    y:B<R1>
```

for `x.f=y` to be type-safe, we require `R?=R1`

# Region Type Checking - Modus Operandi

```
class B {}   ⟶   class B<ρ0> { }

class A {    ⟶   class A<ρ0,ρ1|ρ1≽ρ0> {
  B f;             B<ρ1> f;
}                }
```

```
newAlloc {
  var x = new A();
  newAlloc {
    var y = new B();
    x.f = y;
  }
  print (x.f);
}
```

# Region Type Checking - Modus Operandi

```
class B {}          →   class B<ρ0> { }
```

```
class A {           →   class A<ρ0,ρ1|ρ1≽ρ0> {
  B f;                    B<ρ1> f;
}                       }
```

```
newAlloc {                      → newAlloc<R0>
  var y = new B();              →   new B<R0>()
  newAlloc {                    → newAlloc<R1>
    var x = new A();            →   new A<R1,R0>()
    x.f = y;
  }
  print (x.f);
}
```

$\underbrace{\texttt{x.f : B<R0>} \quad \texttt{y:B<R0>}}$

for `x.f=y` is now perfectly safe

We have looked at examples with non-transferable static regions.

The type system prevented dangling references by only allowing references to longer living regions.

But, does this approach work for transferable regions?

# Broom Regions - Example

```
void select(Region<Batch<Record>> rin,
                          Func<Record,OutRec> udf) {
    var rtrans = new Region<Batch<OutRecord>>();
    open (rin) {
        var batch = rin.GetData();
        openAlloc (rtrans) {
            rtrans.SetData(new Batch<OutRecord>());
        }
        newAlloc {
            foreach var x in batch {
                var y = udf(x);
                openAlloc (rtrans) {
                    var z = rtrans.SuckObject(y);
                    rtrans.GetData().append(y);
                }
            }
        }
    }
    rin.GiveUp();
    rtrans.Transfer(this.getDownStreamActor());
}
```

y is alloc'ed in static region R

y copied to rtrans

Its lifetime ends here

possible dangling ptr dereference in downstream actor

# Broom Regions - Example

```
void select(Region<Batch<Record>> rin,
                      Func<Record,OutRec> udf) {
    var rtrans = new Region<Batch<OutRecord>>();
    open (rin) {
        var batch = rin.GetData();
        openAlloc (rtrans) {
            rtrans.SetData(new Batch<OutRecord>());
        }
        newAlloc {
          foreach var x in batch {
              var y = udf(x);
              openAlloc (rtrans) {
                  var z = rtrans.SuckObject(y);
                  rtrans.GetData().append(y);
              }
          }
        }
    }
    rin.GiveUp();
    rtrans.Transfer(this.getDownStreamActor());
}
```

`newAlloc<R0>`

`y:OutRec<R0>`

If we assume $R0 \succcurlyeq rtrans$ then typecheck passes!

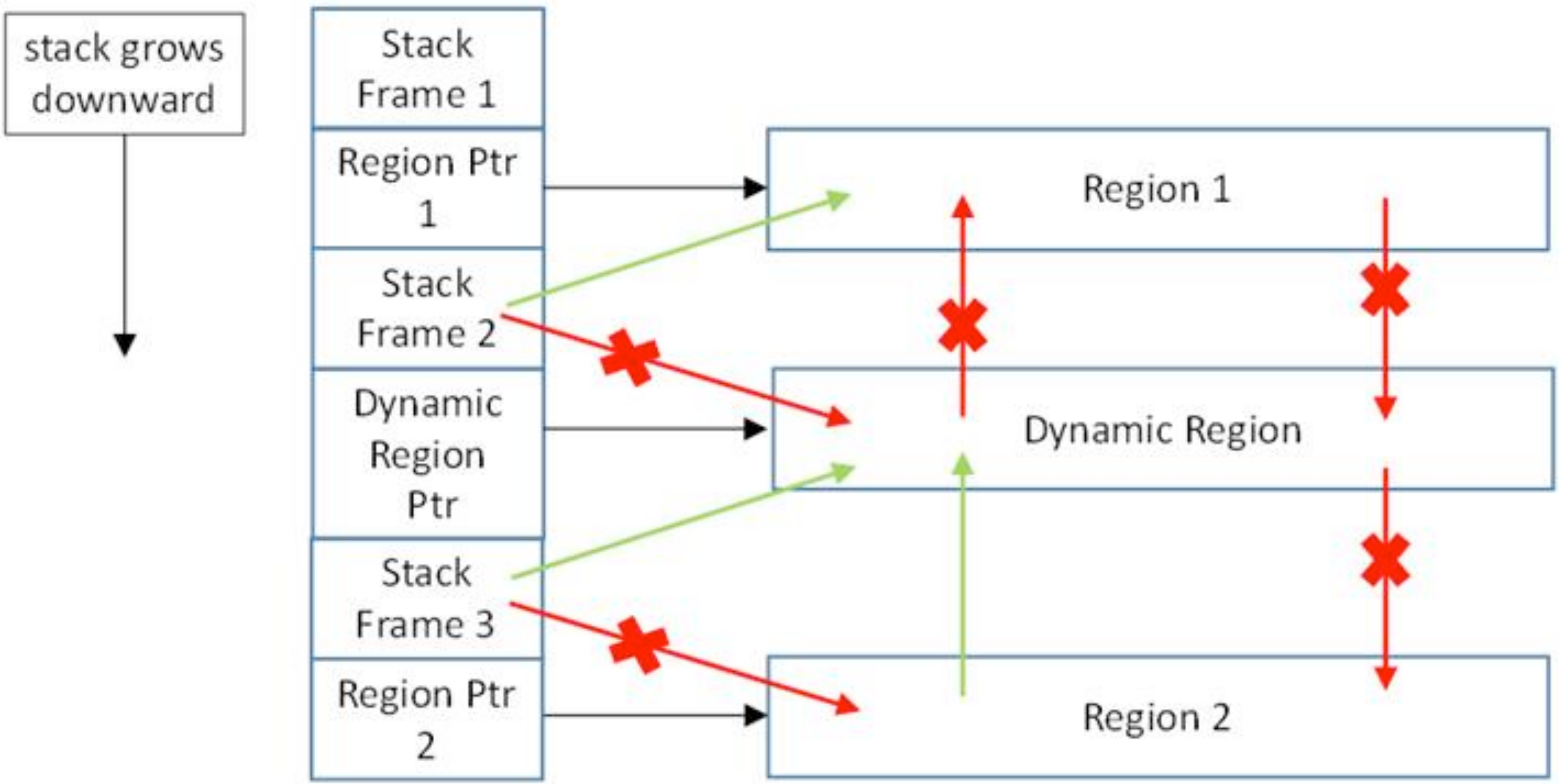
possible dangling ptr dereference in downstream actor

# Observation

A sufficient condition to ensure the absence of dangling references from a transferable region is to ensure the absence of *any* references from the transferable region to other regions.

$\Leftrightarrow$

No region outlives a transferable region


stack grows downward
Stack Frame 1
Region Ptr 1
Stack Frame 2
Dynamic Region Ptr
Stack Frame 3
Region Ptr 2
Region 1
Dynamic Region
Region 2

# Broom Regions - Example

```
void select(Region<Batch<Record>> rin,
                          Func<Record,OutRec> udf) {
    var rtrans = new Region<Batch<OutRecord>>();
    open (rin) {
        var batch = rin.GetData();
        openAlloc (rtrans) {
            rtrans.SetData(new Batch<OutRecord>());
        }
        newAlloc {                  newAlloc<R0>
            foreach var x in batch {
                var y = udf(x);     y:OutRec<R0>
                openAlloc (rtrans) {
                    var z = rtrans.SuckObject(y);
                    rtrans.GetData().append(y);
                }                                 Now fails typecheck
            }
        }
    }
    rin.GiveUp();
    rtrans.Transfer(this.getDownStreamActor());
}
```

possible dangling ptr dereference in downstream actor

# Region Type Inference

Region type checking requires annotating `newAlloc’`s and class names with region names (eg: `newAlloc<R0>, A<R1,R0>, B<R0>` etc)

Can these annotations be inferred from plain C# types?

# Region Type Inference

# Region Type Inference - Intuitions

```
class LinkedListNode<T extends Object> extends
Object  {
    T val;
    LinkedListNode<T>  prev;
    LinkedListNode<T>  next;
    LinkedListNode(T val) {
        super();
        this.val = val;
        this.prev = Null;
        this.head = Null;
    }
}
```

# Region Type Inference - Intuitions

```
class LinkedListNode<T extends Object<ρ1>> extends
Object<ρ0>  {
    T val;
    LinkedListNode<T>  prev;
    LinkedListNode<T>  next;
    LinkedListNode(T val) {
        super();
        this.val = val;
        this.prev = Null;
        this.head = Null;
    }
}
```

# Region Type Inference - Intuitions

```
class LinkedListNode<ρ0,ρ1><T extends Object<ρ1>>
extends Object<ρ0>  {
    T val;
    LinkedListNode<T>  prev;
    LinkedListNode<T>  next;
    LinkedListNode(T val) {
        super();
        this.val = val;
        this.prev = Null;
        this.head = Null;
    }
}
```

# Region Type Inference - Intuitions

```
class LinkedListNode<ρ0,ρ1><T extends Object<ρ1>>
extends Object<ρ0>  {
    T val;
    LinkedListNode<ρ0,ρ1><T>  prev;
    LinkedListNode<ρ0,ρ1><T>  next;
    LinkedListNode(T val) {
        super();
        this.val = val;
        this.prev = Null;
        this.head = Null;
    }
}
```

# Region Type System

Not covered in today's talk!

- Region type inference -
  - Constraint generation
  - Constraint solving
  - Residual constraints
- Existential typing to assign type to a linked list of transferable regions.
- Why current type system still can't ensure safety with transferable regions.
  - Dynamic solution: throwing exceptions
- InAllocationContext and region-polymorphic types for functions.
- Higher-Order functions.

# Region Type System

## Open problems

- Region type inference in presence of higher-order functions
- Tracing the failure of constraint satisfaction during type inference back to the erroneous instruction in source program.
- Full region inference: Is it possible to take the code that is written assuming GC, and automatically transform it to a better performing code that uses Broom regions?

# Region Type System

## Theorems yet to be proved

- Type safety - If a broom program has a type under the region type system, then the program does not dereference a pointer into a dead region*
- Soundness of type inference - If type inference elaborates (transforms) a program p to program p′, then program p′ is well-typed under region type system.

*Conditions Apply

Thank you!