Effect Analysis for Programs with Callbacks

Etienne Kneuss, Viktor Kuncak, Philippe Suter

VSTTE 2013 – 17.05.2013

```
class Cell {
  var visited = false
}
def toggle(c: Cell) {
  c.visited = !c.visited
}
```

What does this function do?

```
class Cell {
  var visited = false
}
def toggle(c: Cell) {
  c.visited = !c.visited
}
```

What does this function do?

“Updates the field ***c.visited****”*

```
class Cell {
  var visited = false
}
def toggle(c: Cell) {
  c.visited = !c.visited
}
def apply(c: Cell, f: Cell=>Unit) {
  f(c)
}
```

What does this function do?

*"Updates the field **c.visited**"*

```
class Cell {
  var visited = false
}
def toggle(c: Cell) {
  c.visited = !c.visited
}
def apply(c: Cell, f: Cell=>Unit) {
  f(c)
}
```

What does this function do?

*“Updates the field **c.visited**”*

*“Calls function **f** on cell **c**”*

```
class Cell {
  var visited = false
}
def toggle(c: Cell) {
  c.visited = !c.visited
}
def apply(c: Cell, f: Cell=>Unit) {
  f(c)
}
def visitAll(cs: List[Cell]) {
   cs.foreach( _.visited = true )
}
```

What does this function do?

*“Updates the field **c.visited**”*

*“Calls function **f** on cell **c**”*

```
class Cell {
  var visited = false
}
def toggle(c: Cell) {
  c.visited = !c.visited
}
def apply(c: Cell, f: Cell=>Unit) {
  f(c)
}
def visitAll(cs: List[Cell]) {
   cs.foreach( _.visited = true )
}
```

What does this function do?

*"Updates the field **c.visited**"*

*"Calls function **f** on cell **c**"*

*"Sets the **visited** field on all cells of **cs**"*

Motivation

- Goal: Precise effect analysis
  - Important for automated reasoning
  - Enables e.g. compiler optimizations
  - *Additional goal*: improve program understanding
- Challenges:
  - Functions cannot be analyzed in isolation
  - Naïve approaches fall short on dynamic dispatch
  - Analysis results can be hard to interpret

Contributions

- A precise pointer and effect analysis

  flow-sensitive, modular, supports higher-order functions, requires no annotations

- A translation of effects to readable summaries

```
def visitAll(cs: List[Cell]) {
   cs.foreach{ _.visited = true }
}
```

Contributions

- A precise pointer and effect analysis

  flow-sensitive, modular, supports higher-order functions, requires no annotations

- A translation of effects to readable summaries

```
def visitAll(cs: List[Cell]) {
    cs.foreach{ _.visited = true }
}
```

cs.tl*.hd.visited

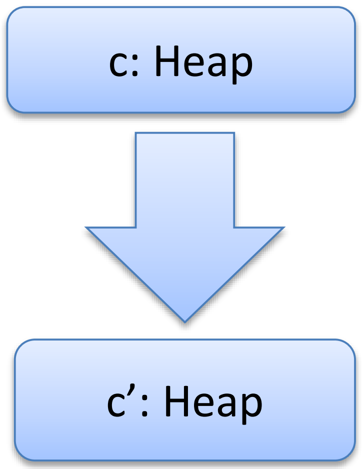
Relational Analysis

Concrete

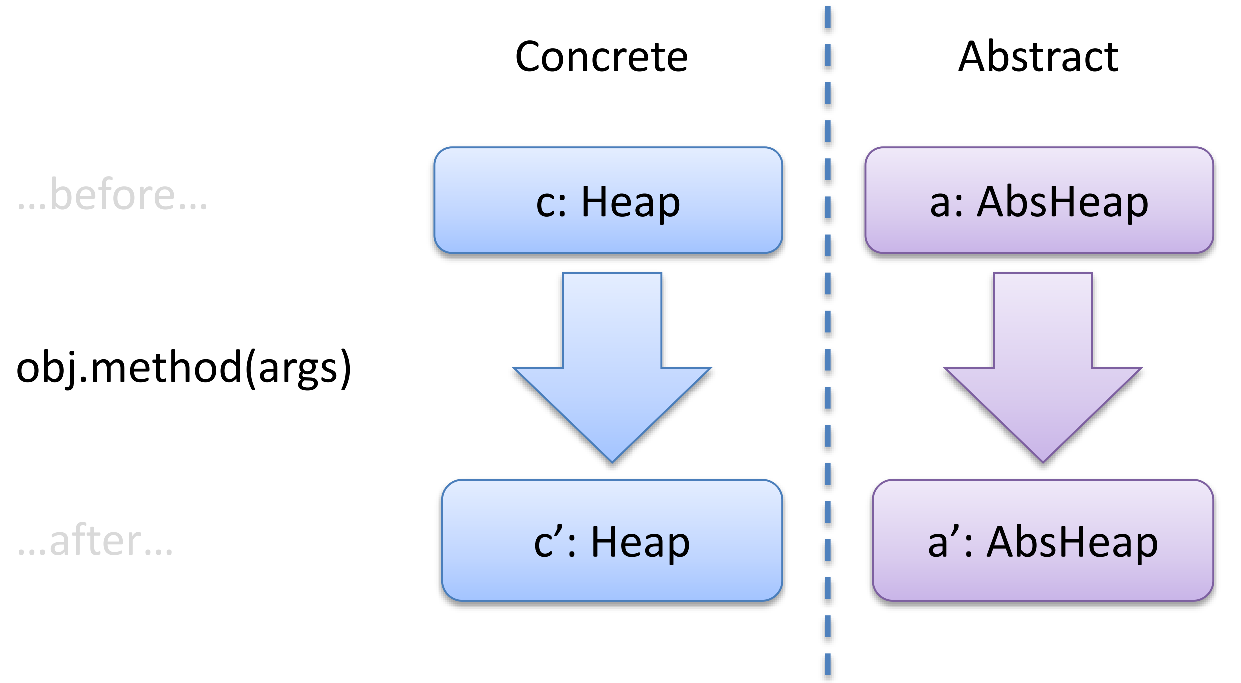
…before…

obj.method(args)

…after…

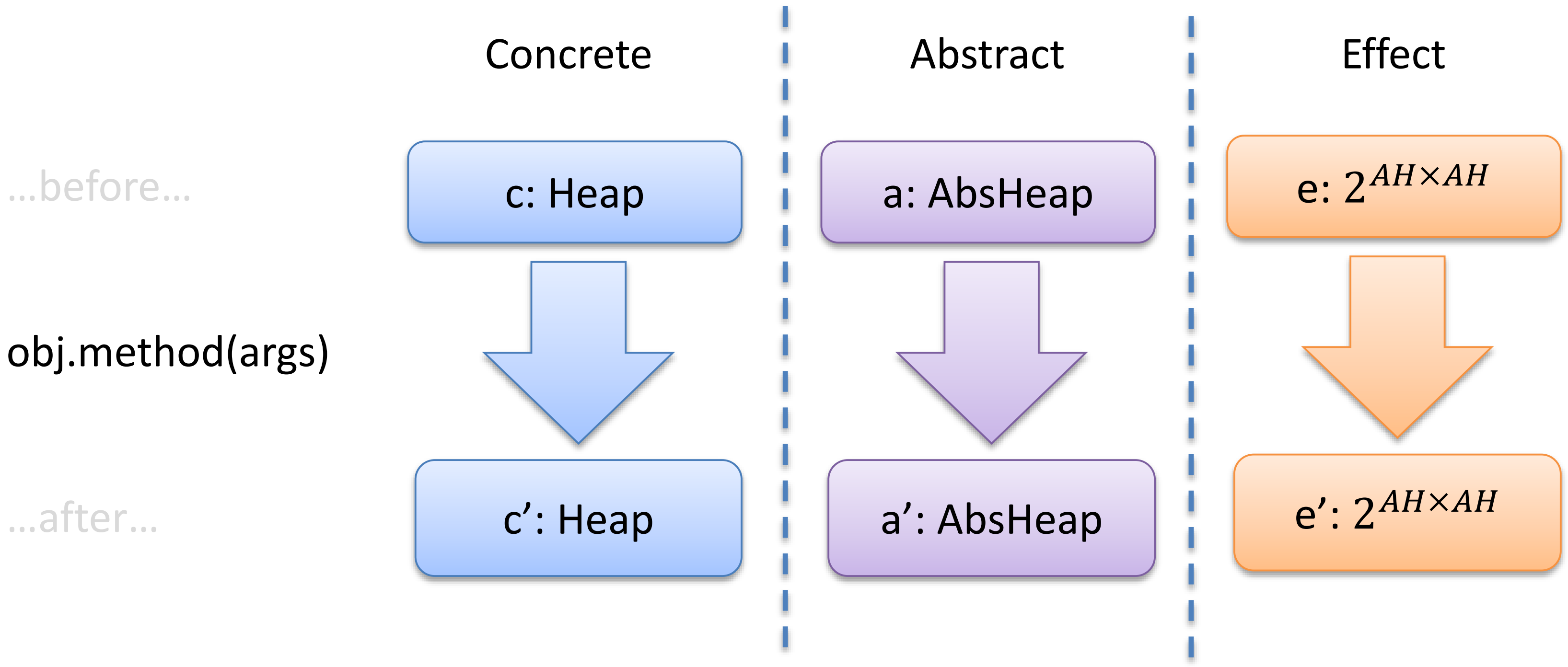


Relational Analysis



E.g. points-to

Relational Analysis



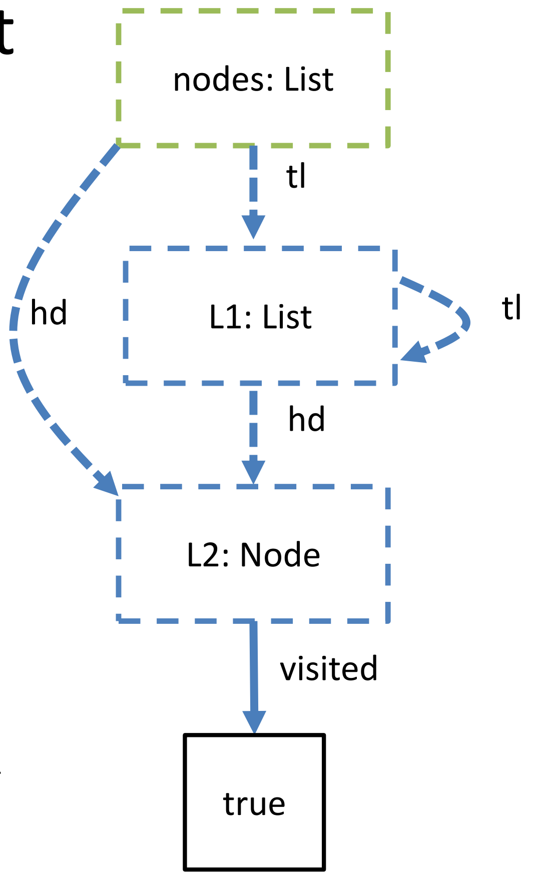
E.g. points-to

Effects as Graphs

- Graphs describe relations on abstract heaps:

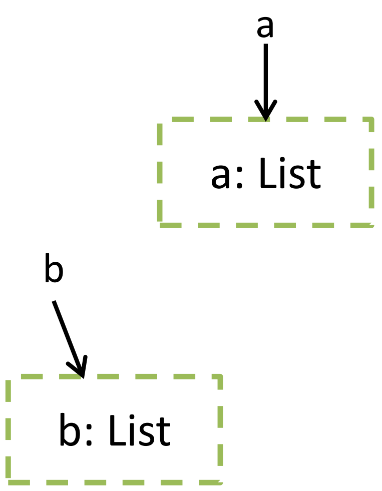
$$E: 2^{AH \times AH}$$

- Nodes represent objects.
- Edges encode read or write effects.
- Nodes may be unresolved:
  - parameters, fields, **this**
- Domain adapted from previous work by Salcianu et al.



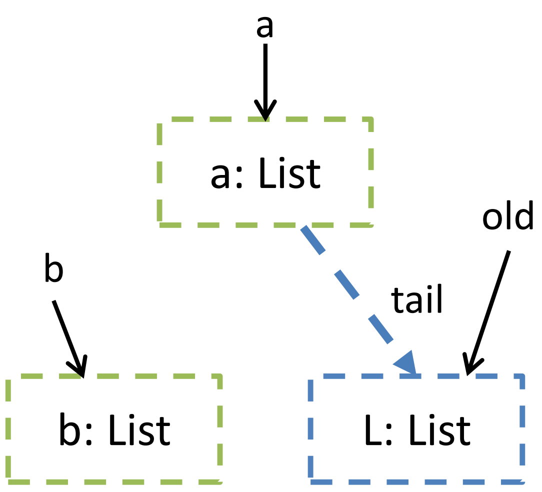
Example

```
case class List(var head: Int,
                var tail: List)

def setTail(a: List, b: List): List = {
    val old = a.tail
    a.tail = b
    old
}
```



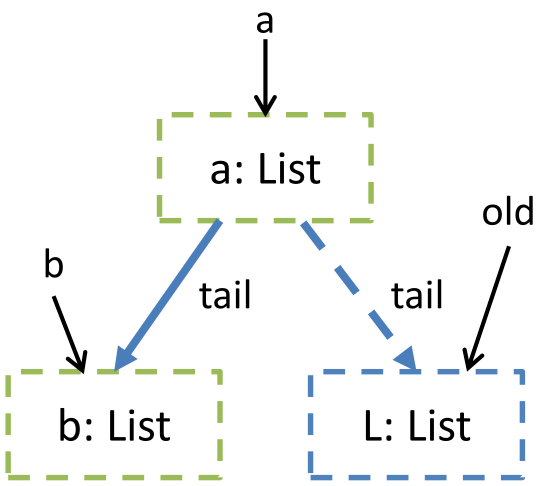
Example

```
case class List(var head: Int,
                var tail: List)

def setTail(a: List, b: List): List = {
    val old = a.tail
    a.tail = b
    old
}
```



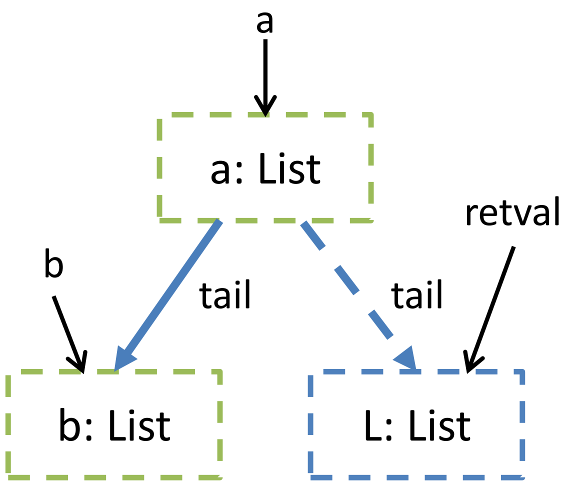
Example

```
case class List(var head: Int,
                var tail: List)

def setTail(a: List, b: List): List = {
  val old = a.tail
  a.tail = b
  old
}
```



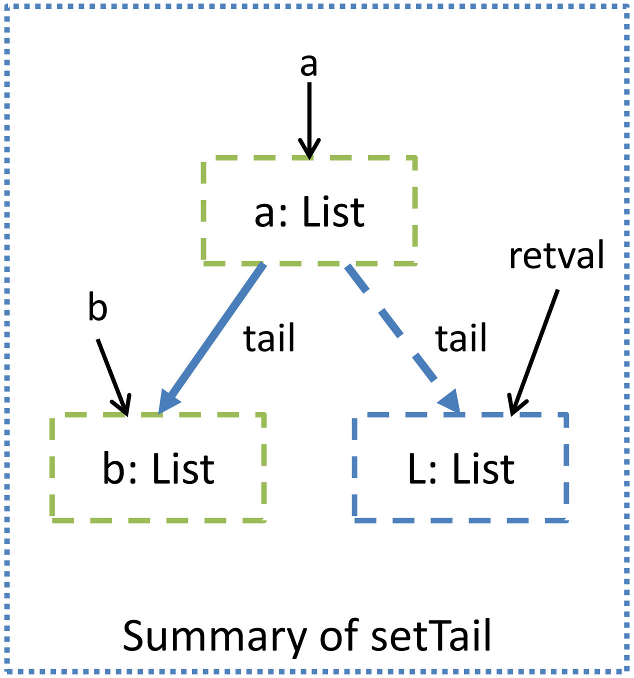
Example

```
case class List(var head: Int,
                var tail: List)

def setTail(a: List, b: List): List = {
   val old = a.tail
   a.tail = b
   old
}
```



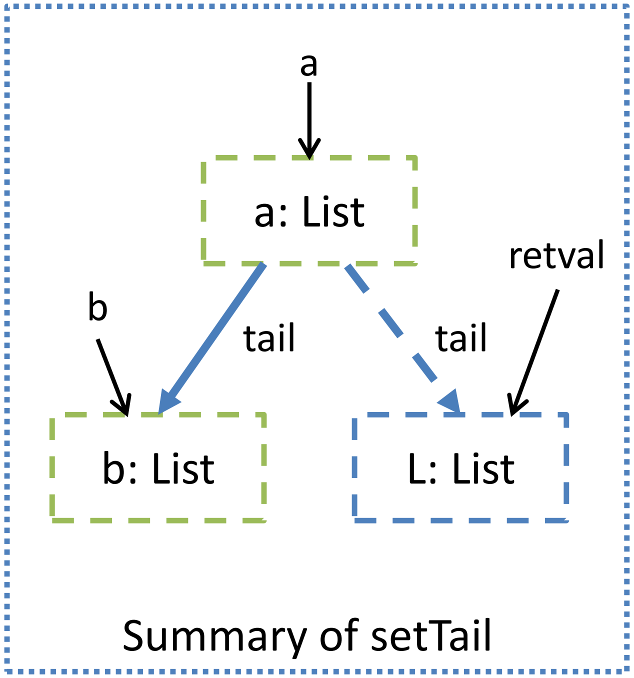
Example

```
case class List(var head: Int,
                var tail: List)

def setTail(a: List, b: List): List = {
   val old = a.tail
   a.tail = b
   old
}
```



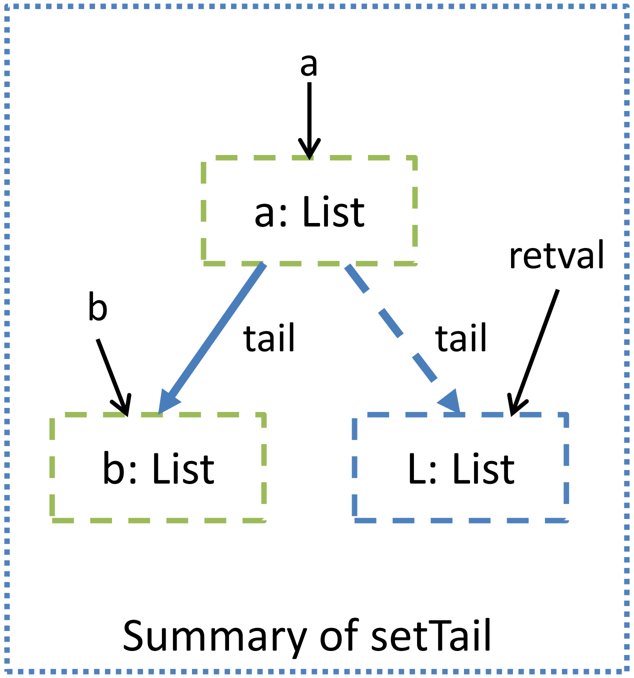
Example

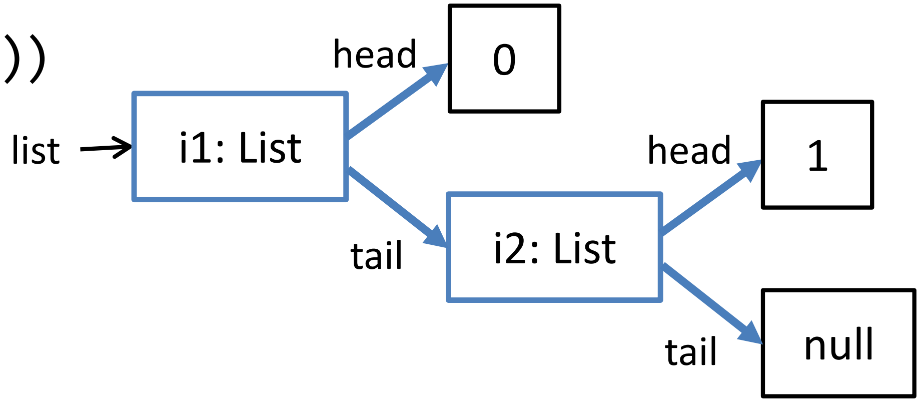
```
case class List(var head: Int,
                var tail: List)


def setTail(a: List, b: List): List = {
   val old = a.tail
   a.tail = b
   old
}

def example() {
   val list = new List(0,
                   new List(1, null))
   setTail(list, list)
}
```



Example

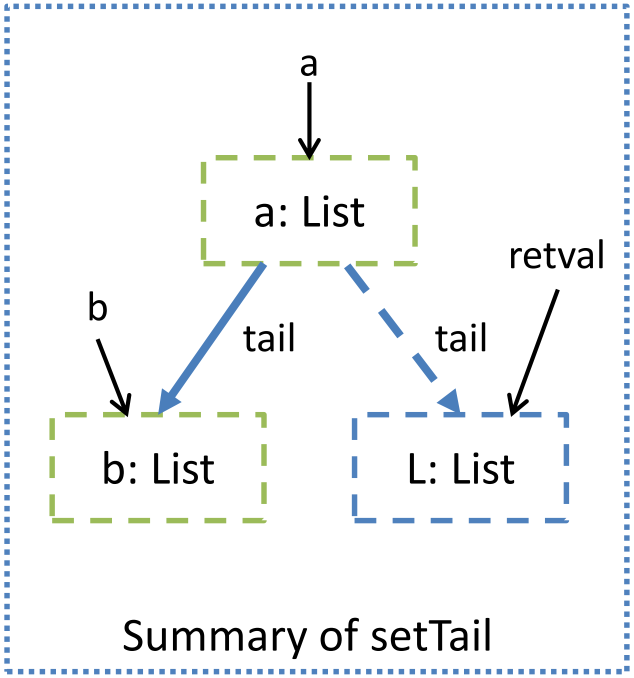
```
case class List(var head: Int,
                var tail: List)


def setTail(a: List, b: List): List = {
    val old = a.tail
    a.tail = b
    old
}

def example() {
    val list = new List(0,
                   new List(1, null))
→   setTail(list, list)
}
```

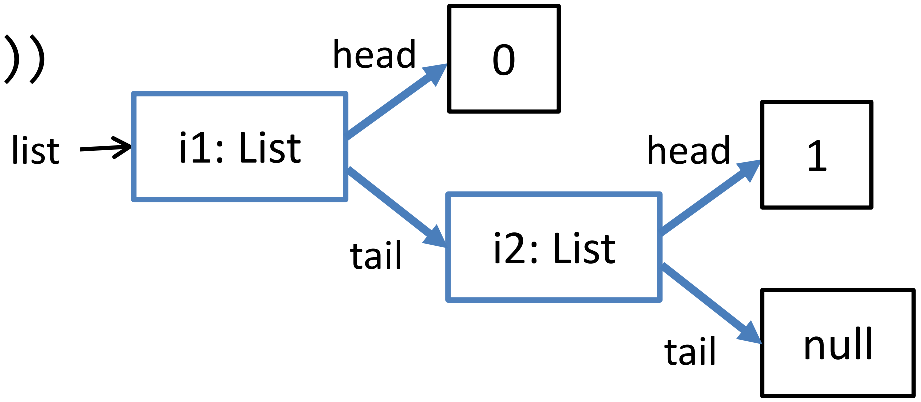




Example

```
case class List(var head: Int,
                var tail: List)


def setTail(a: List, b: List): List = {
   val old = a.tail
   a.tail = b
   old
}

def example() {
   val list = new List(0,
                 new List(1, null))
   setTail(list, list)
}
```

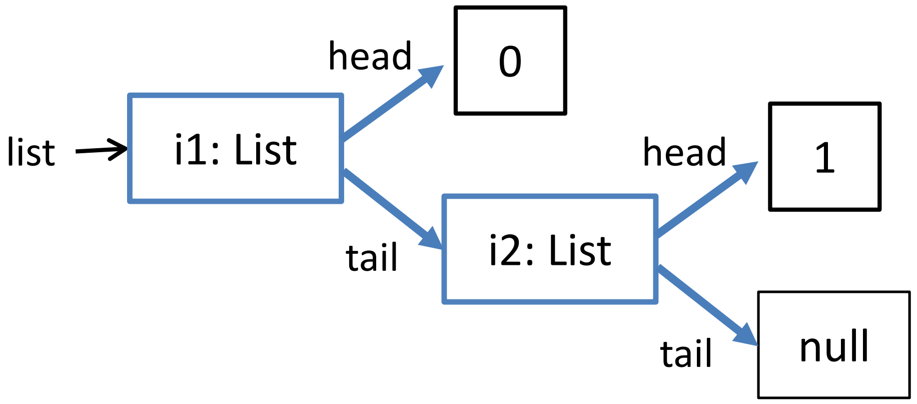


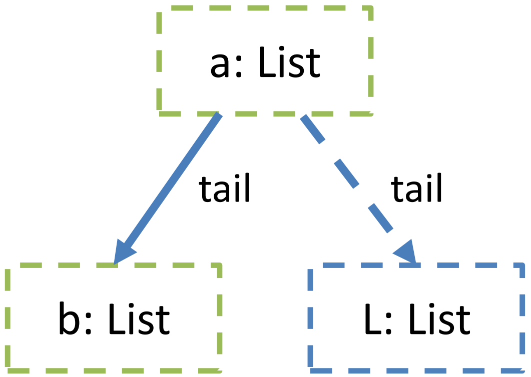


?

Composition

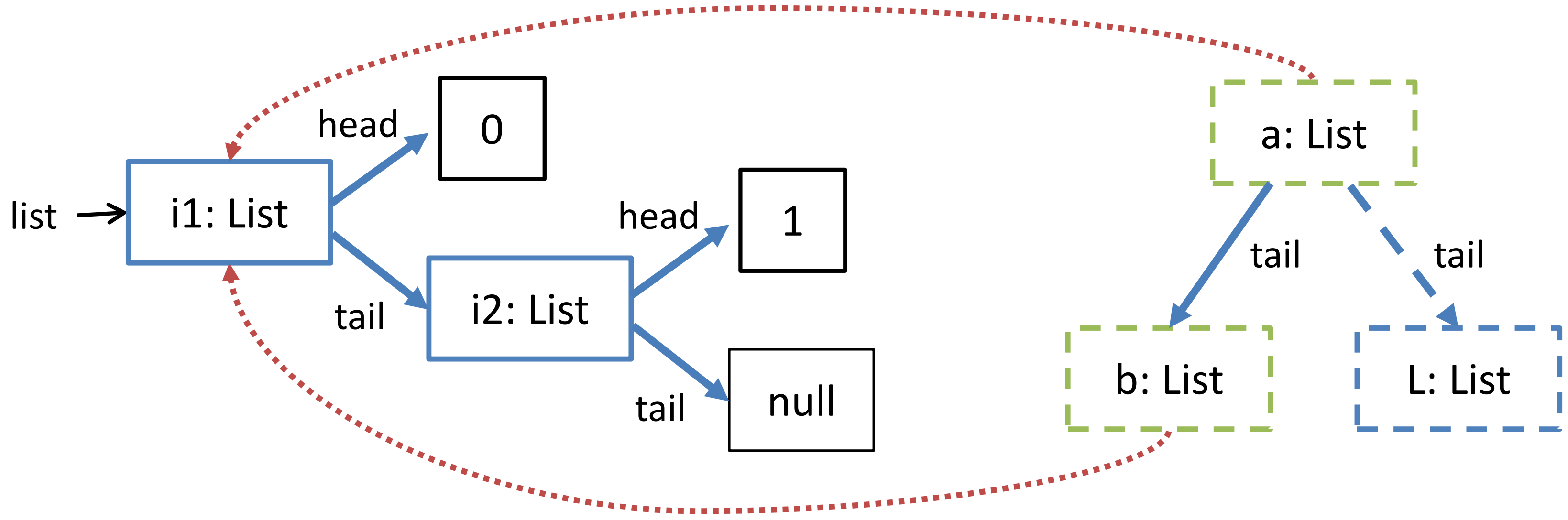
```
→ map parameters to arguments;
  do:
    resolve nodes;
    import write effects;
  until fix-point;
```





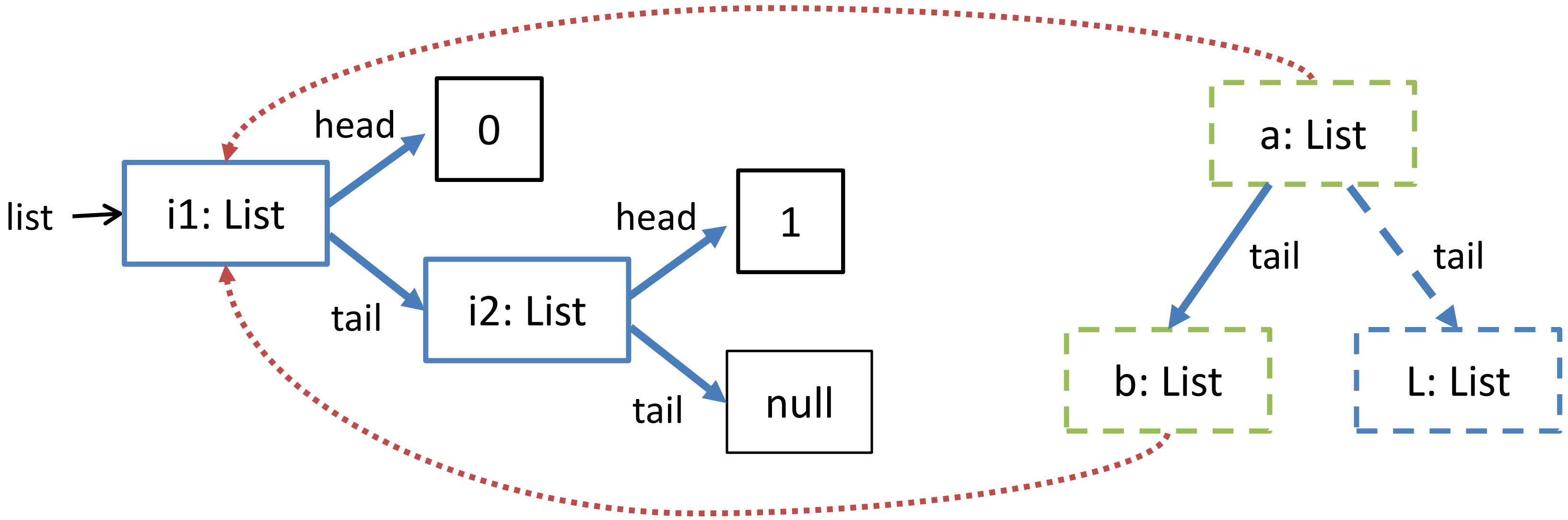
Composition

```
→ map parameters to arguments;
  **do**:
    resolve nodes;
    import write effects;
  **until** fix-point;
```



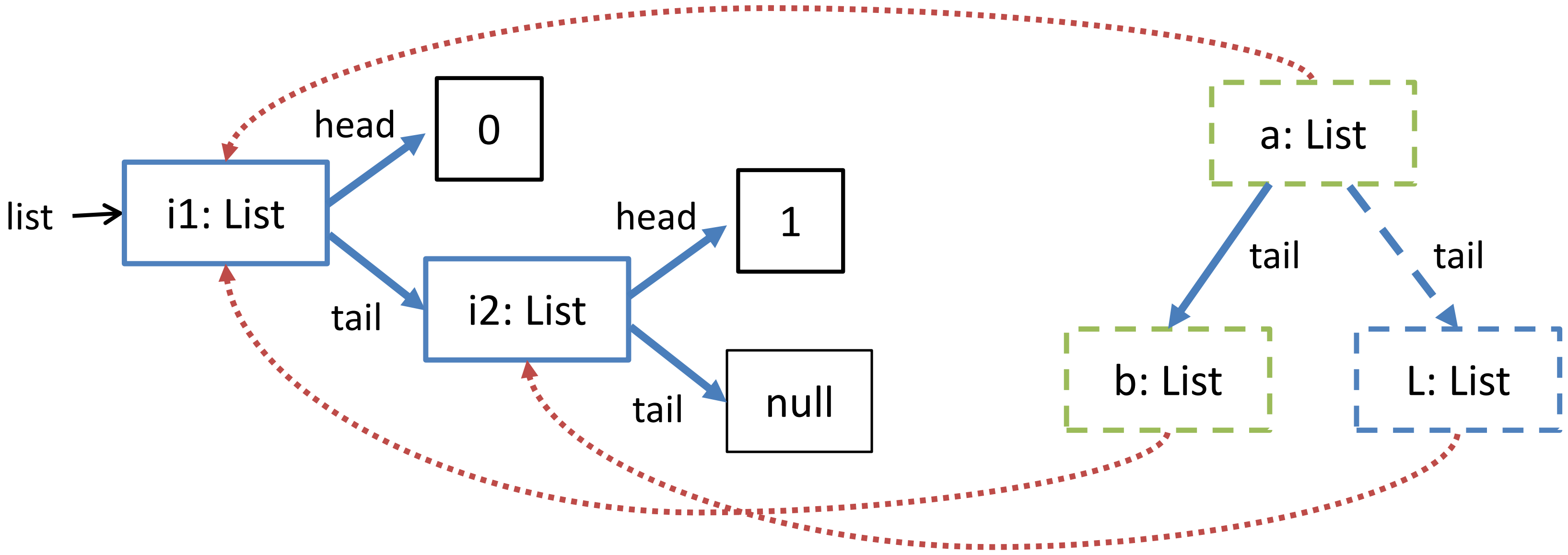
Composition

```
map parameters to arguments;
**do**:
→ resolve nodes;
  import write effects;
**until** fix-point;
```



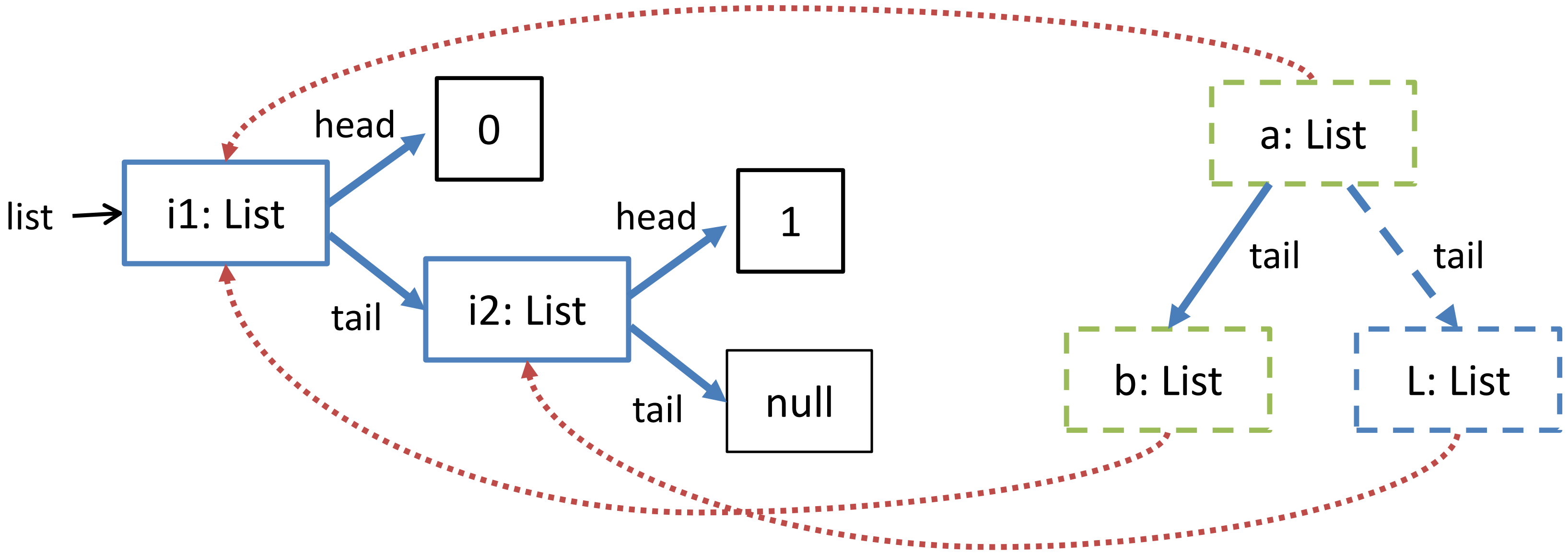
Composition

```
map parameters to arguments;
do:
→ resolve nodes;
  import write effects;
until fix-point;
```



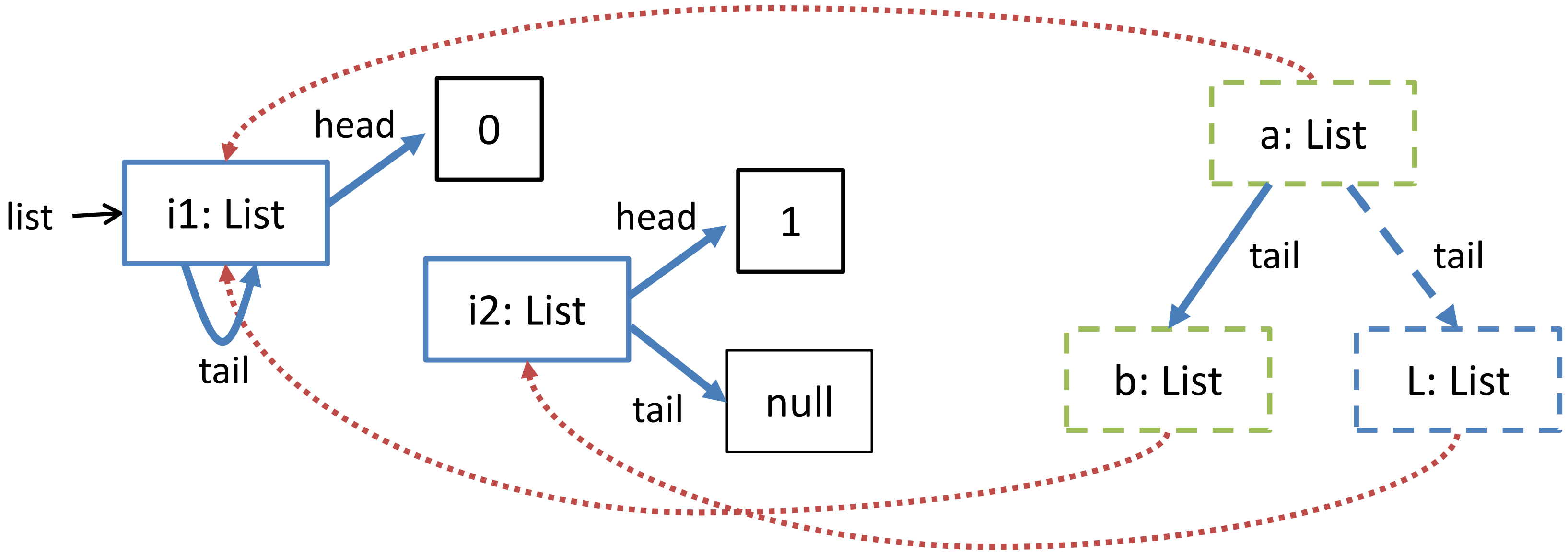
Composition

```
map parameters to arguments;
do:
  resolve nodes;
→ import write effects;
until fix-point;
```



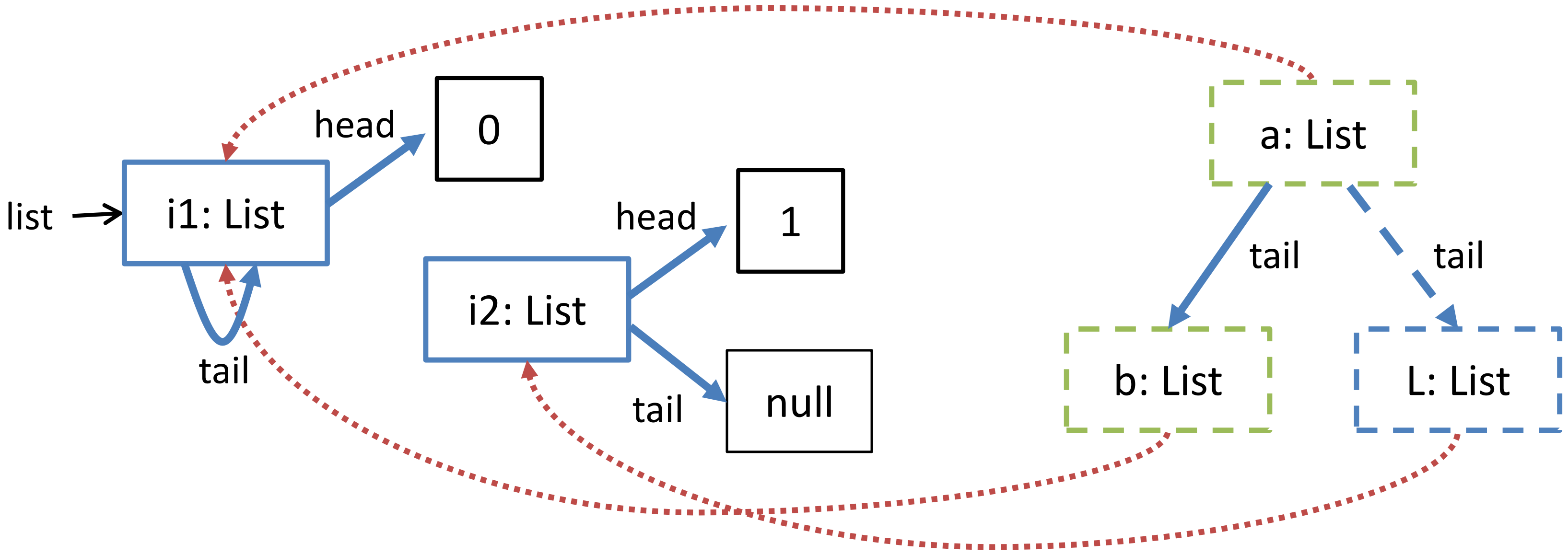
Composition

```
map parameters to arguments;
do:
  resolve nodes;
→ import write effects;
until fix-point;
```



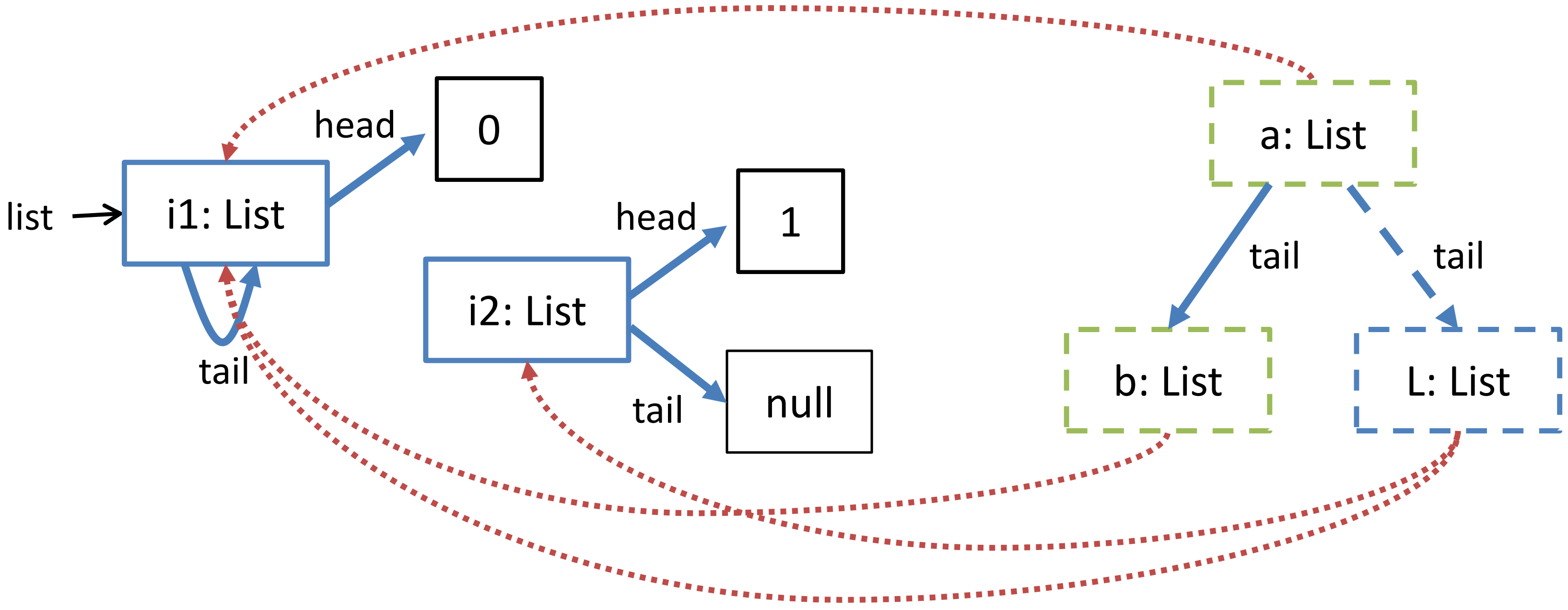
Composition

```
map parameters to arguments;
do:
→ resolve nodes;
  import write effects;
until fix-point;
```



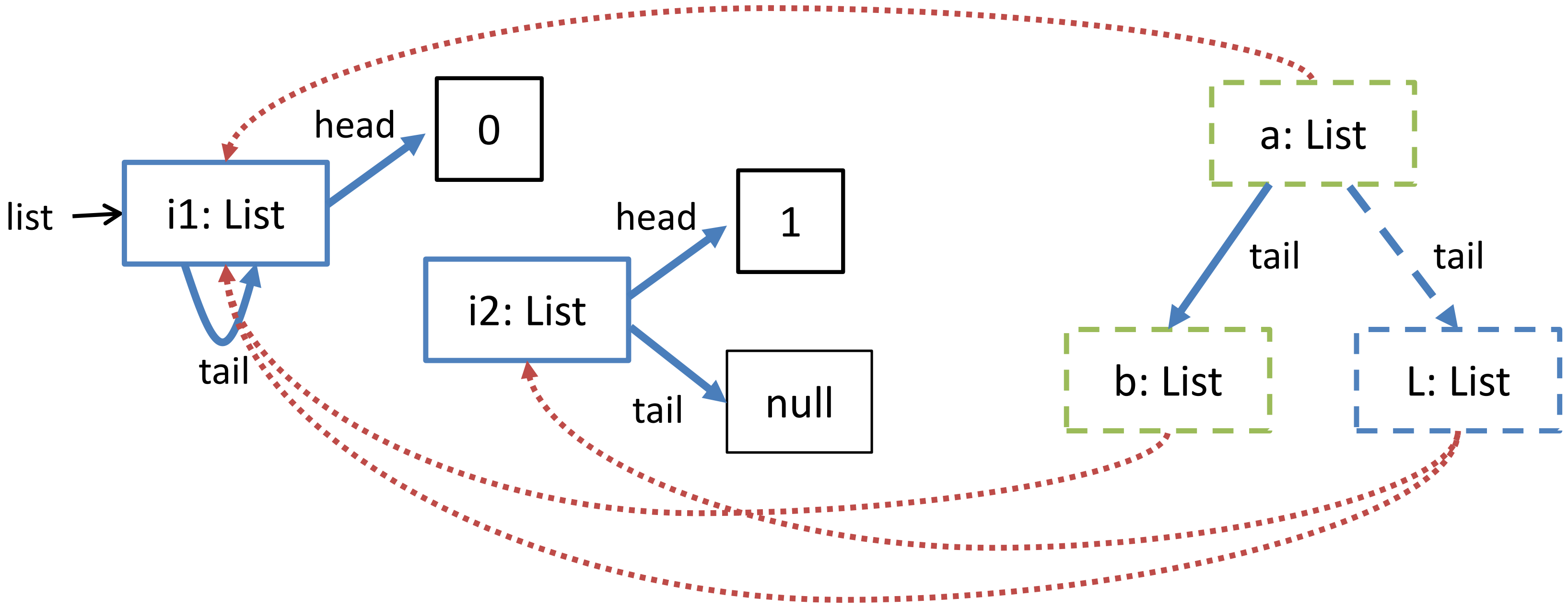
Composition

```
map parameters to arguments;
do:
→ resolve nodes;
  import write effects;
until fix-point;
```



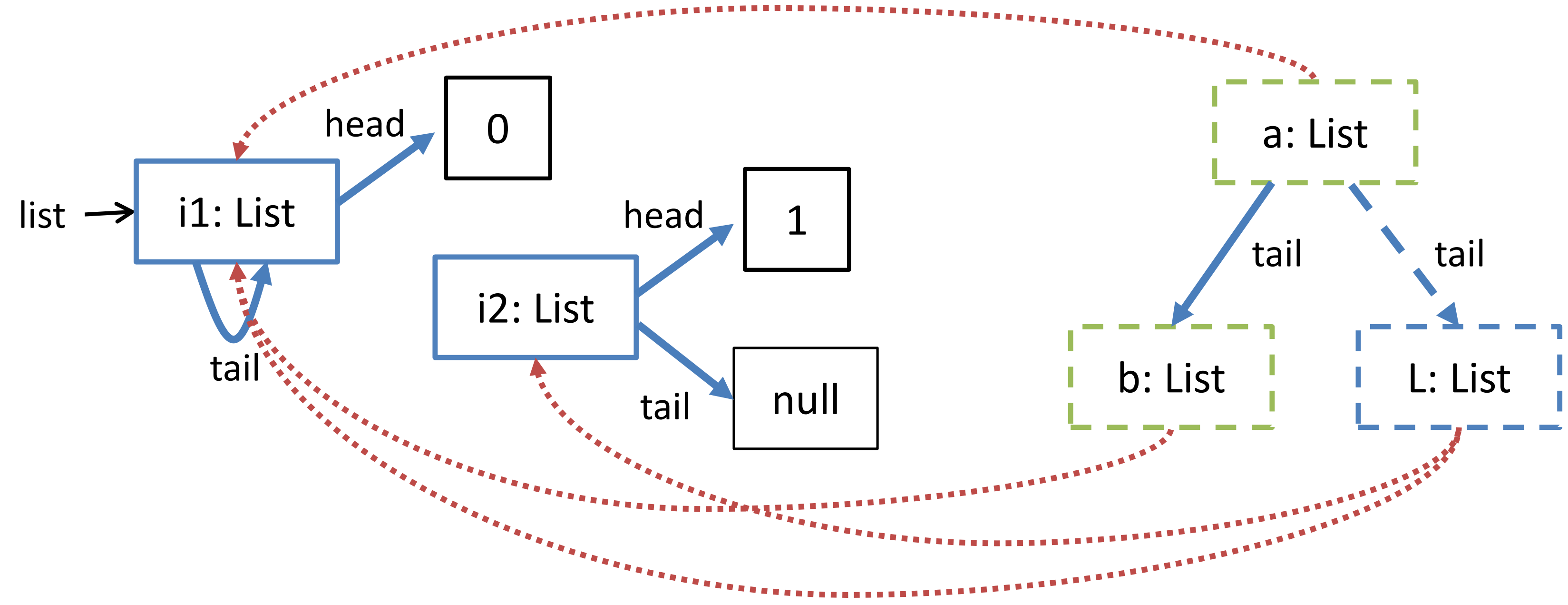
Composition

```
  map parameters to arguments;
  do:
    resolve nodes;
→   import write effects;
  until fix-point;
```



Composition

```
map parameters to arguments;
do:
    resolve nodes;
    import write effects;
until fix-point;
```



Effects as Graphs

- Compact representation of abstract heap transformers.
- Unresolved nodes offer a flexible solution to aliasing problems.
- Composition expressed as a graph manipulation algorithm.

- …but potentially difficult to interpret.
  - best suited as an internal representation

Handling Callbacks

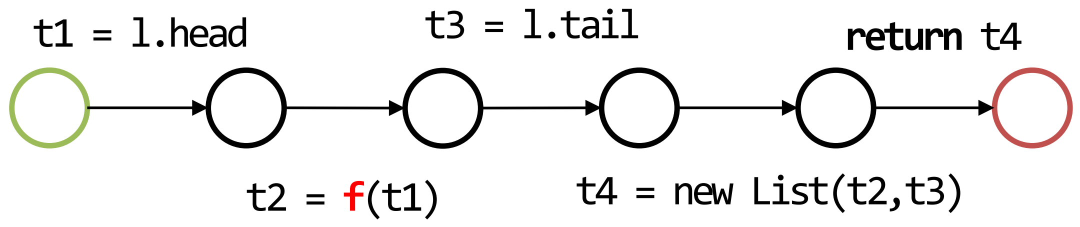
- Dynamic dispatch in practice:
  - Function1 in Scala library has >1000 subclasses
- Union of all potential targets not an option.
- *Idea*: delay analysis of method call until more information is available

Delaying Effect Composition

```
def mapHead(l: List, f: Int=>Int): List = {
  new List(f(l.head), l.tail)
}
```

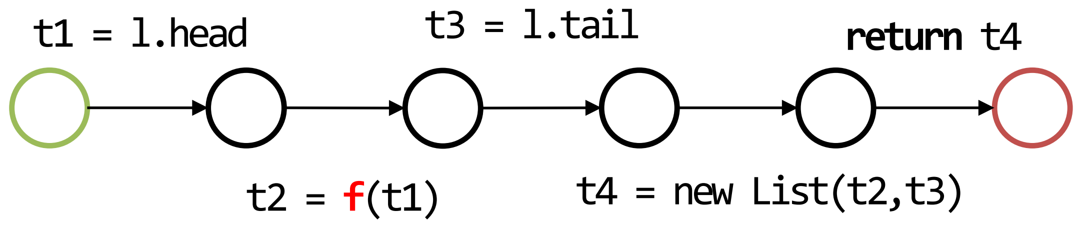
Delaying Effect Composition

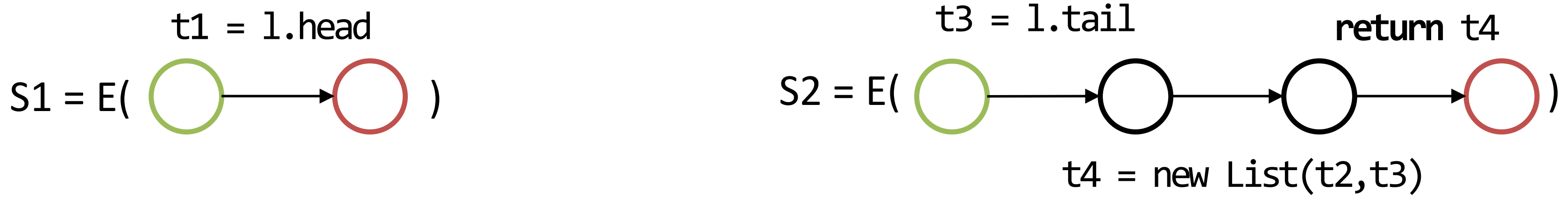
```
def mapHead(l: List, f: Int=>Int): List = {
  new List(f(l.head), l.tail)
}
```



Delaying Effect Composition

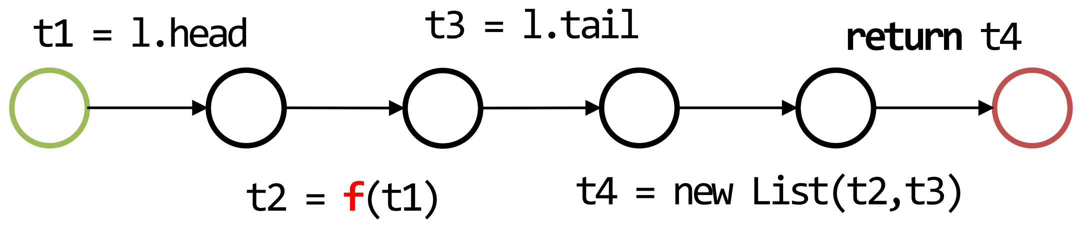
```
def mapHead(l: List, f: Int=>Int): List = {
  new List(f(l.head), l.tail)
}
```

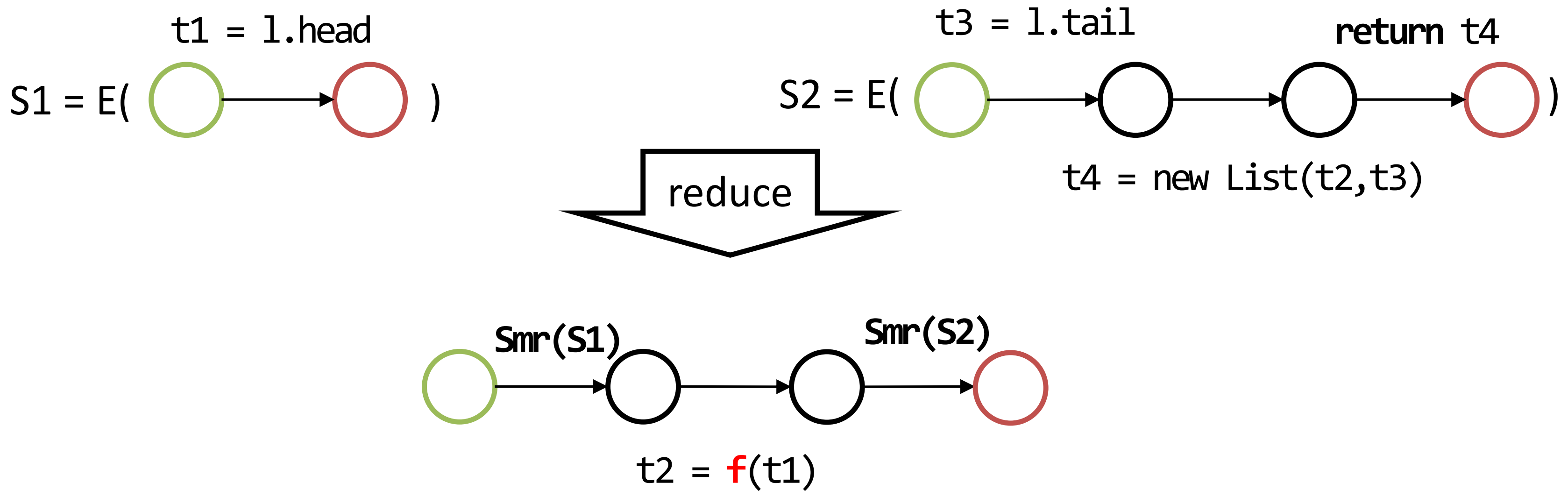




Delaying Effect Composition

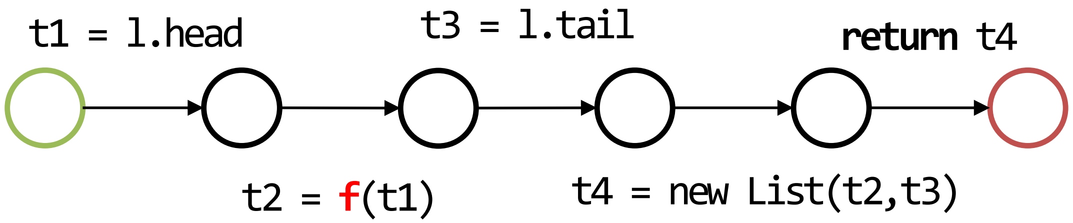
```
def mapHead(l: List, f: Int=>Int): List = {
  new List(f(l.head), l.tail)
}
```

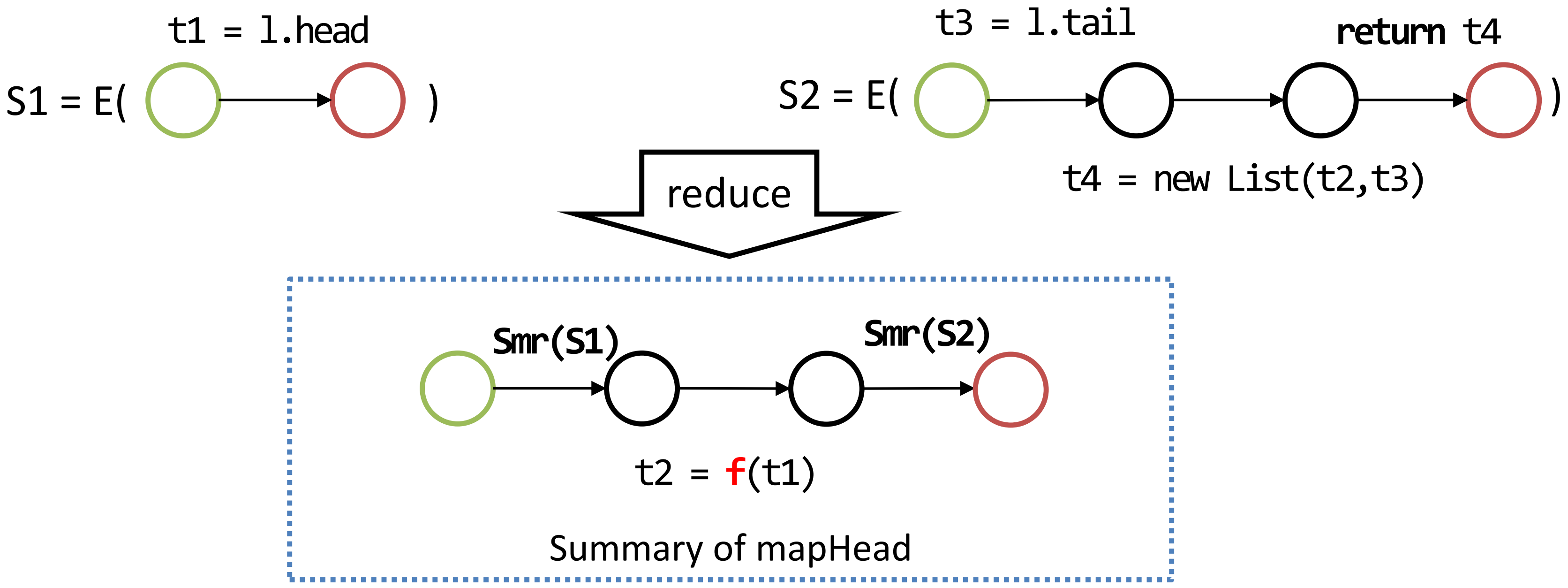




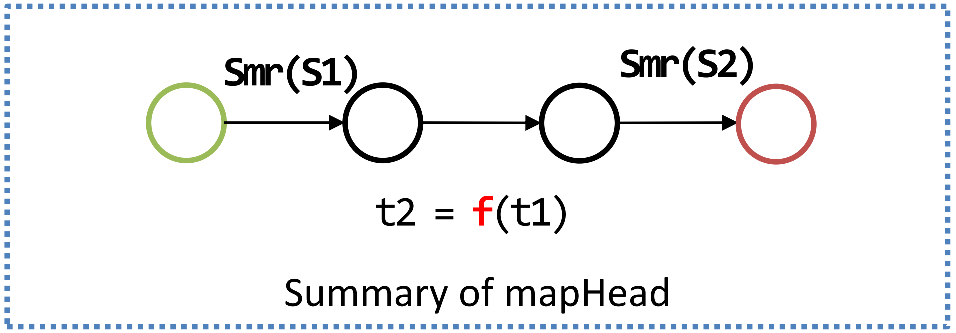
Delaying Effect Composition

```
def mapHead(l: List, f: Int=>Int): List = {
  new List(f(l.head), l.tail)
}
```

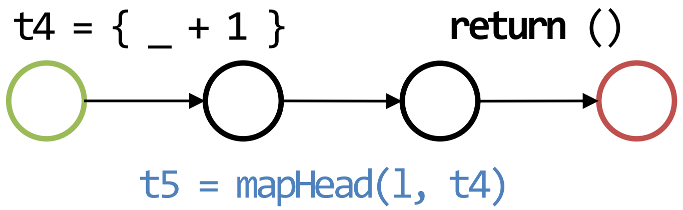




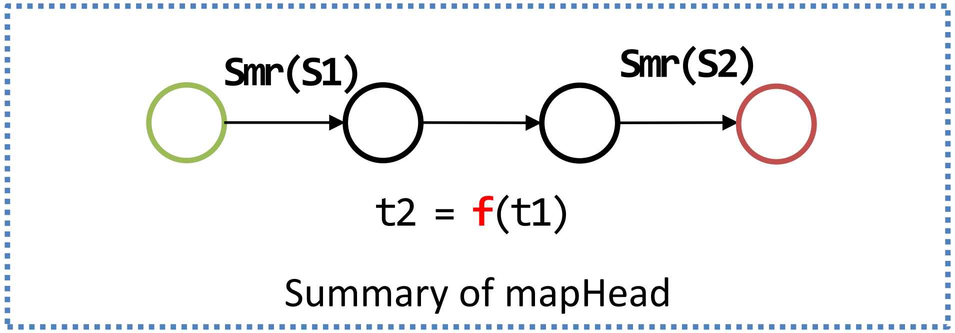
Delaying Effect Composition



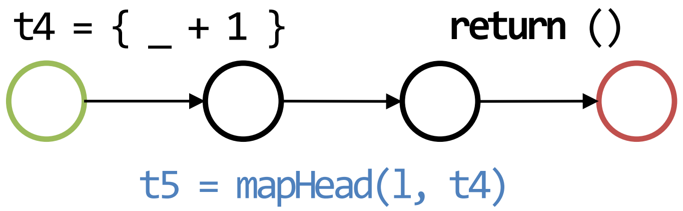
```
def test(l: List) {
   mapHead(l, {_ + 1})
}
```

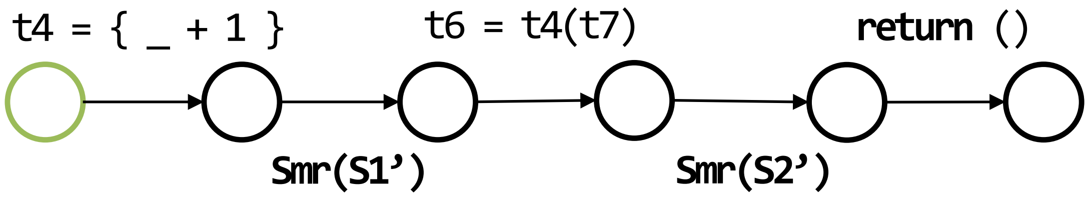


Delaying Effect Composition

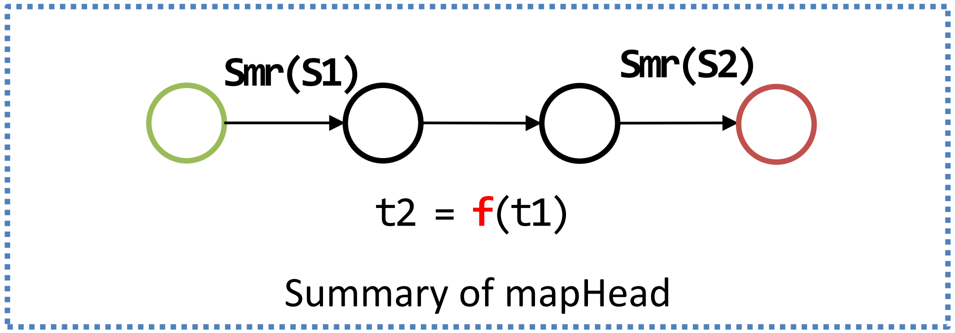


```
def test(l: List) {
    mapHead(l, {_ + 1})
}
```

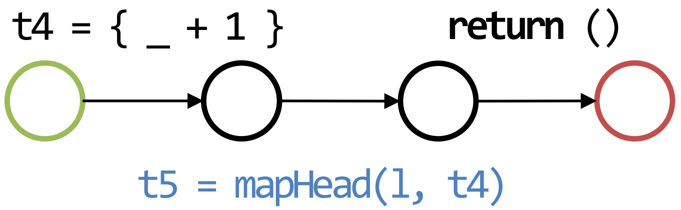


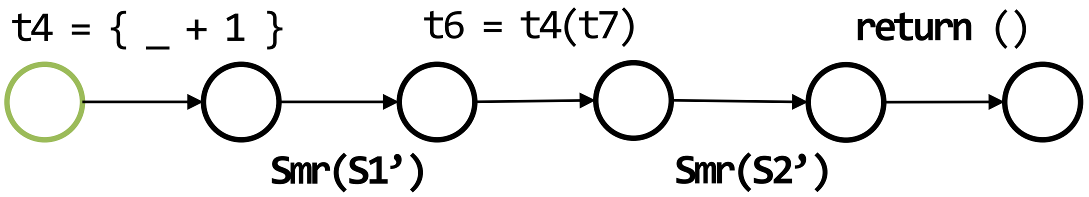


Delaying Effect Composition



```
def test(l: List) {
  mapHead(l, {_ + 1})
}
```

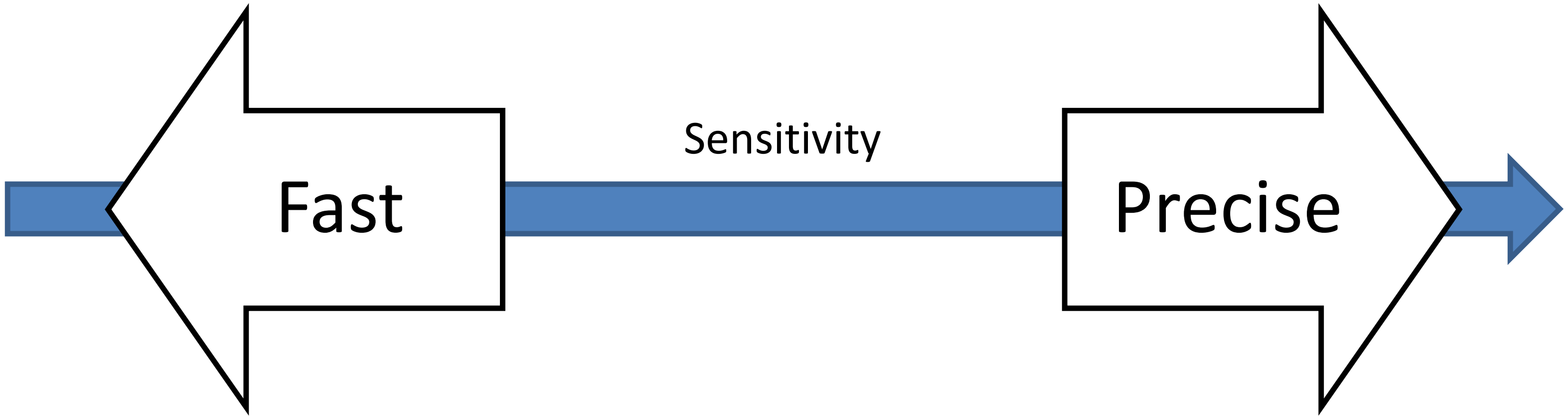




Pure

Delaying Decision

- We base the decision of delaying a method call on several factors:
  - Number of targets
  - Calling context
  - Escaping receiver



insane!

Interprocedural Static Analysis of Effects

- Analysis implemented for Scala
- Plugin for the reference compiler
- Publicly available from:

https://github.com/epfl-lara/insane

Evaluation

- Characterize the effects:

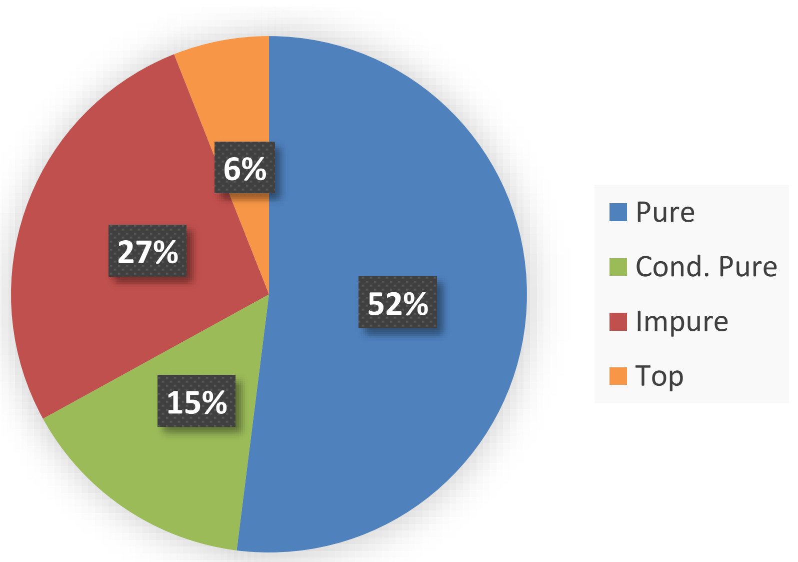
$$pure \sqsubseteq conditionally\ pure \sqsubseteq impure \sqsubseteq \top$$

Evaluation

- Characterize the effects:

$$pure \sqsubseteq conditionally\ pure \sqsubseteq impure \sqsubseteq \top$$

- Run on the Scala 2.10 library:

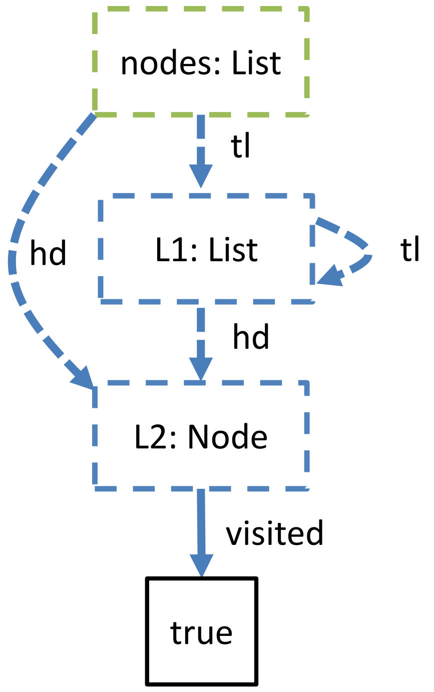


Results

Package	#Methods	Pure	Cond. Pure	Impure	T
scala	5721	79%	11%	10%	1%
scala.annotation	41	93%	2%	2%	2%
scala.beans	25	64%	8%	29%	8%
scala.collection	34810	46%	17%	29%	8%
…	…	…	…	…	…
scala.util	1786	51%	11%	32%	6%
scala.util.parsing	2206	56%	12%	27%	5%
scala.xml	2860	56%	11%	30%	3%
Total	**58410**	**52%**	**15%**	**27%**	**6%**

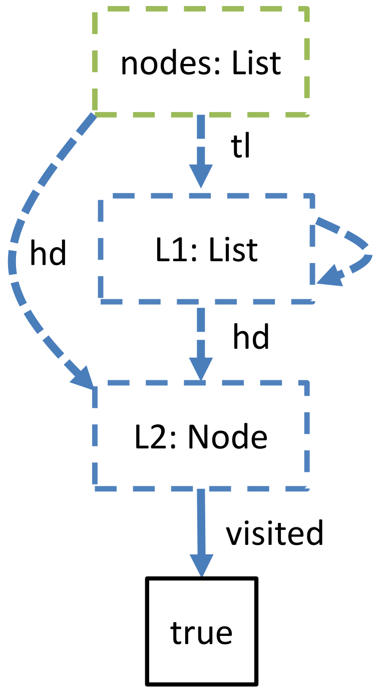
Producing Readable Summaries

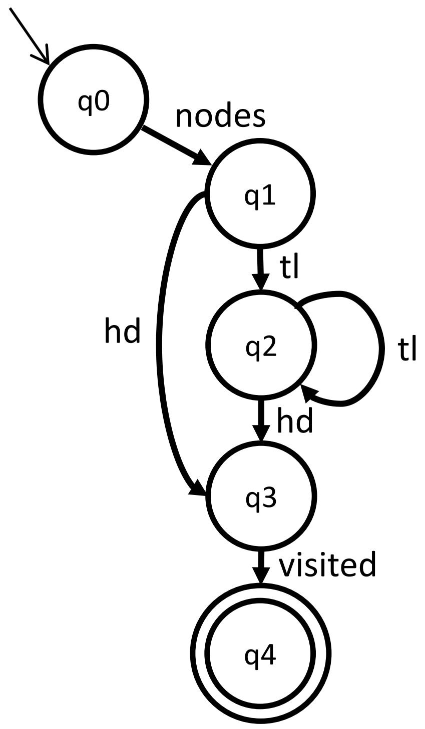
- Translating effect graphs into automata



Producing Readable Summaries

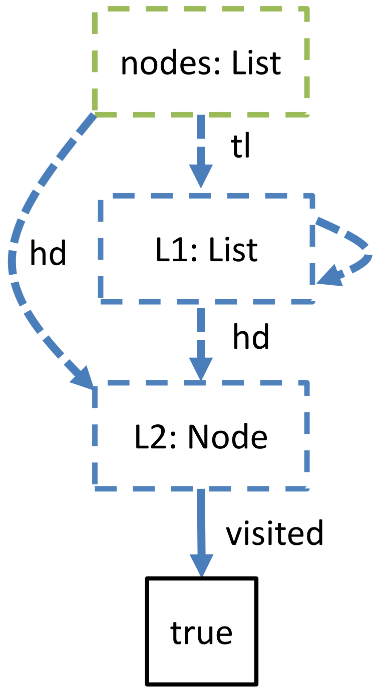
- Translating effect graphs into automata

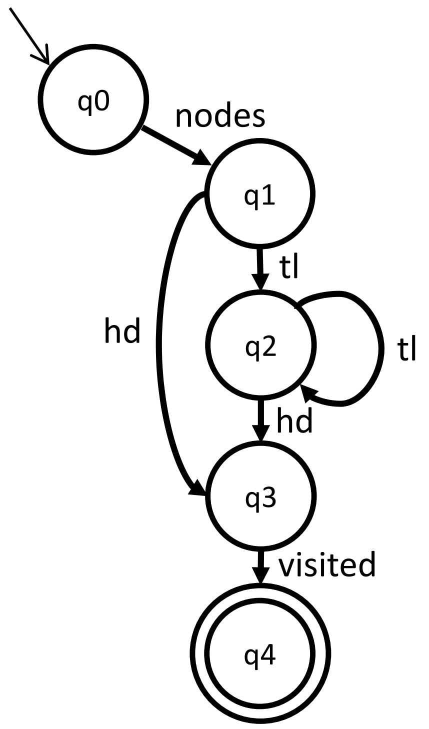


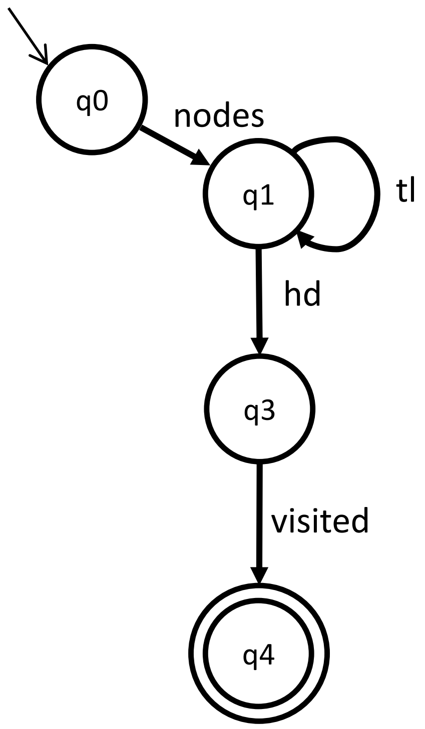


Producing Readable Summaries

- Translating effect graphs into automata

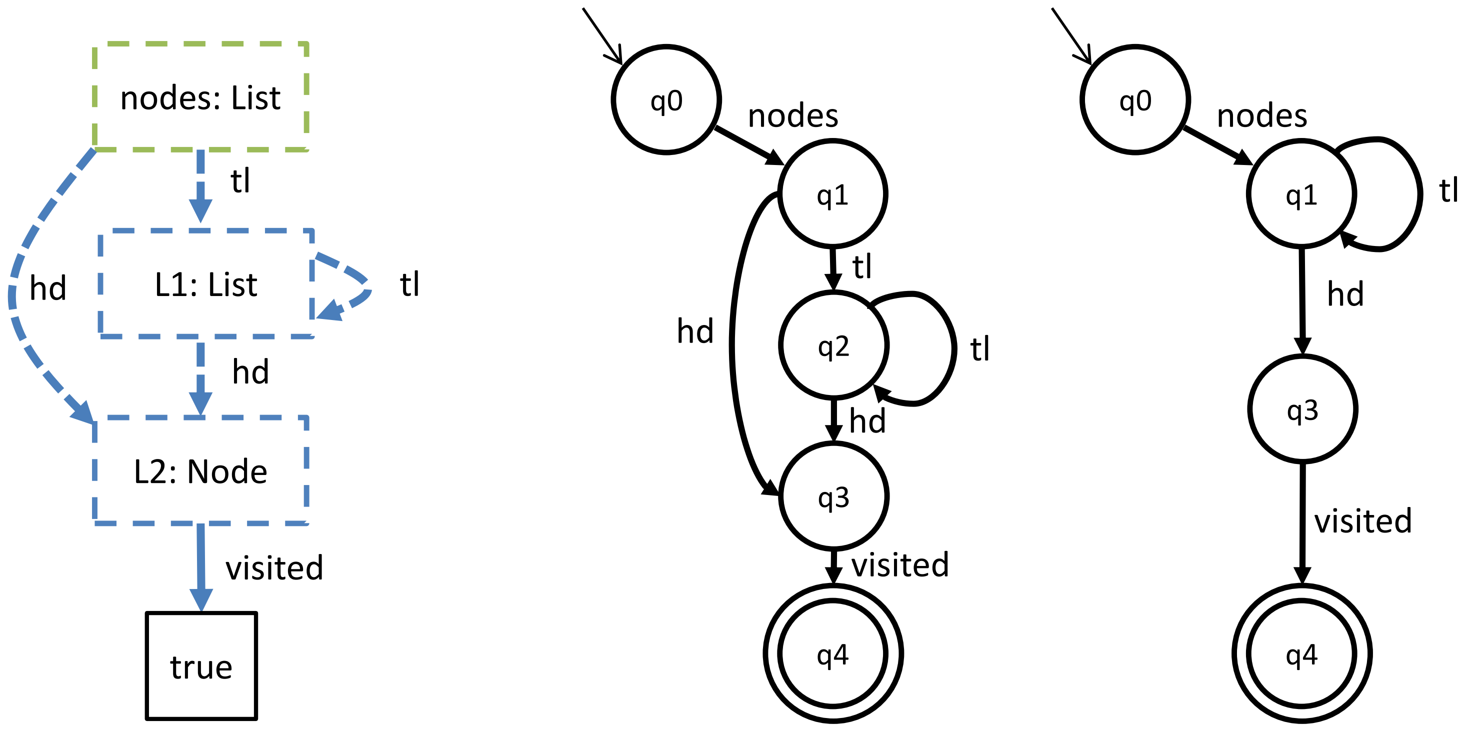






Producing Readable Summaries

- Translating effect graphs into automata



nodes.tl*.hd.visited

Analyzing Collections

Two operations:

Impure Traversal:	`col.foreach{ _.visited = true }`
Grow:	`el => col.append(el)`

On four Scala collections:

- immutable.TreeSet
- immutable.List
- mutable.HashSet
- mutable.LinkedList

Analyzing Collections

- immutable.TreeSet:

Impure Traversal:	es.tree(.right | .left)*.key.visited
Grow:	Pure

- immutable.List:

Impure Traversal:	es.tl*.hd.visited
Grow:	Pure

- mutable.HashSet:

Impure Traversal:	es.table.store.visited
Grow:	es.tableSize | es.table.store | es.sizemap.store | es.sizemap | es.table

- mutable.LinkedList:

Impure Traversal:	es.next*.elem.visited
Grow:	es.next.next*

Selected Related Work

- Salcianu, A.D.: **Pointer Analysis for Java Programs: Novel Techniques and Applications**. Ph.D thesis, MIT (2006)

- Madhavan, R., Ramalingam, G. Waswani, K.: **Modular heap analysis for higher-order programs**. SAS 2012

- Rytz, L., Odersky, M., Haller, P.: **Lightweight polymorphic effects**. ECOOP 2012

Contributions

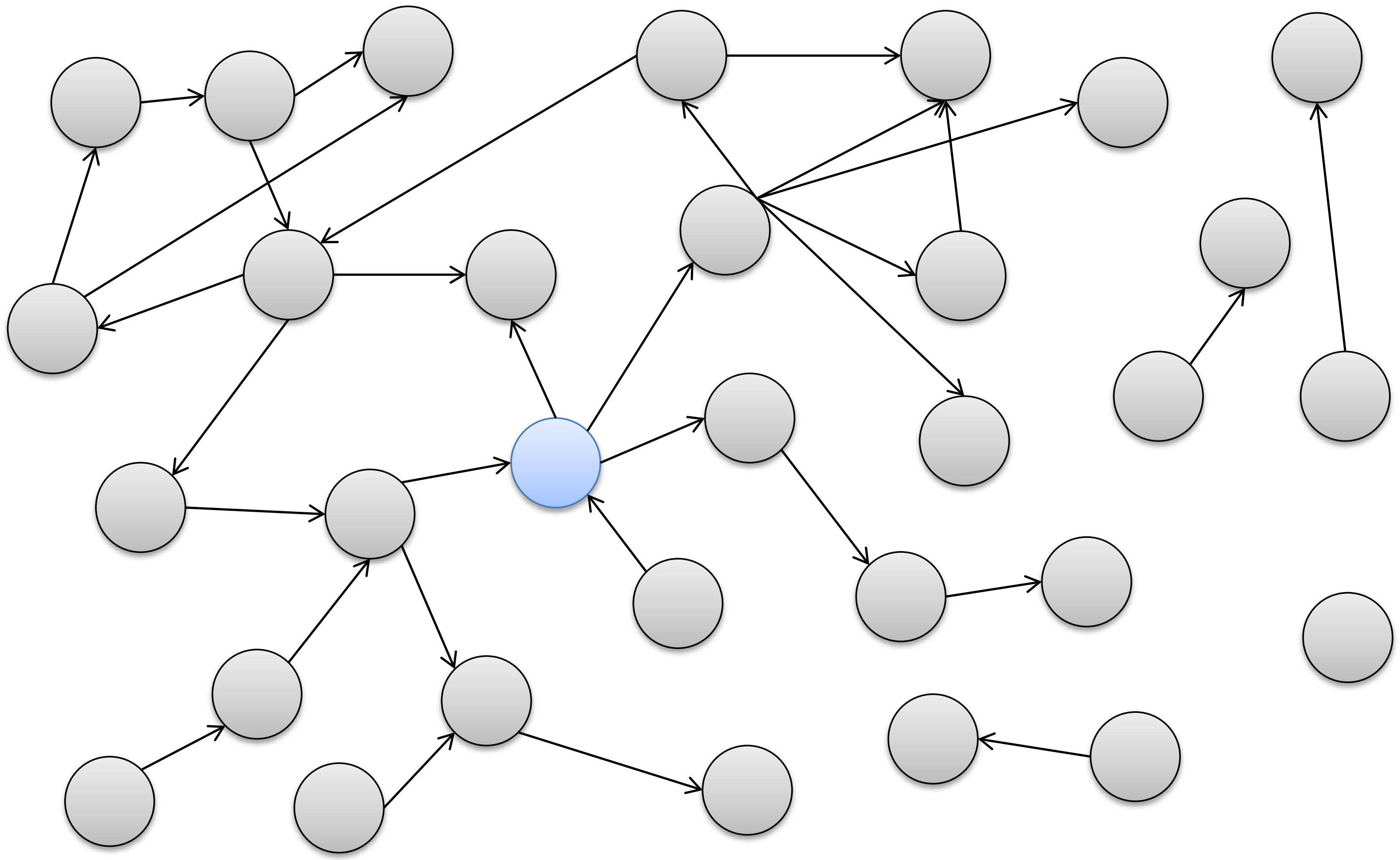
- A precise pointer and effect analysis

  flow-sensitive, modular, supports higher-order functions, requires no annotations
- A translation of effects to readable summaries
- Insane, an analyzer for Scala programs

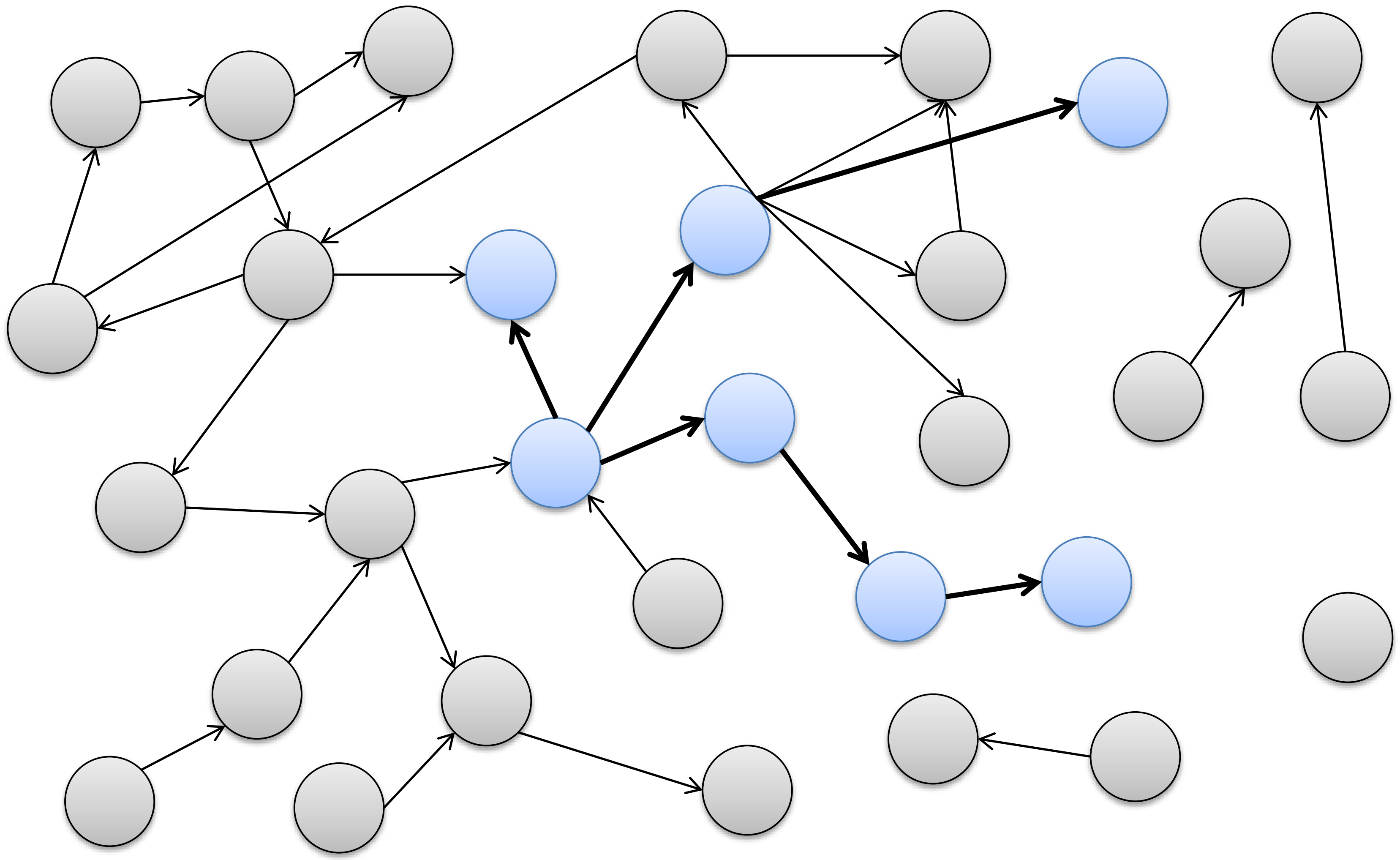
insane!

https://github.com/epfl-lara/insane

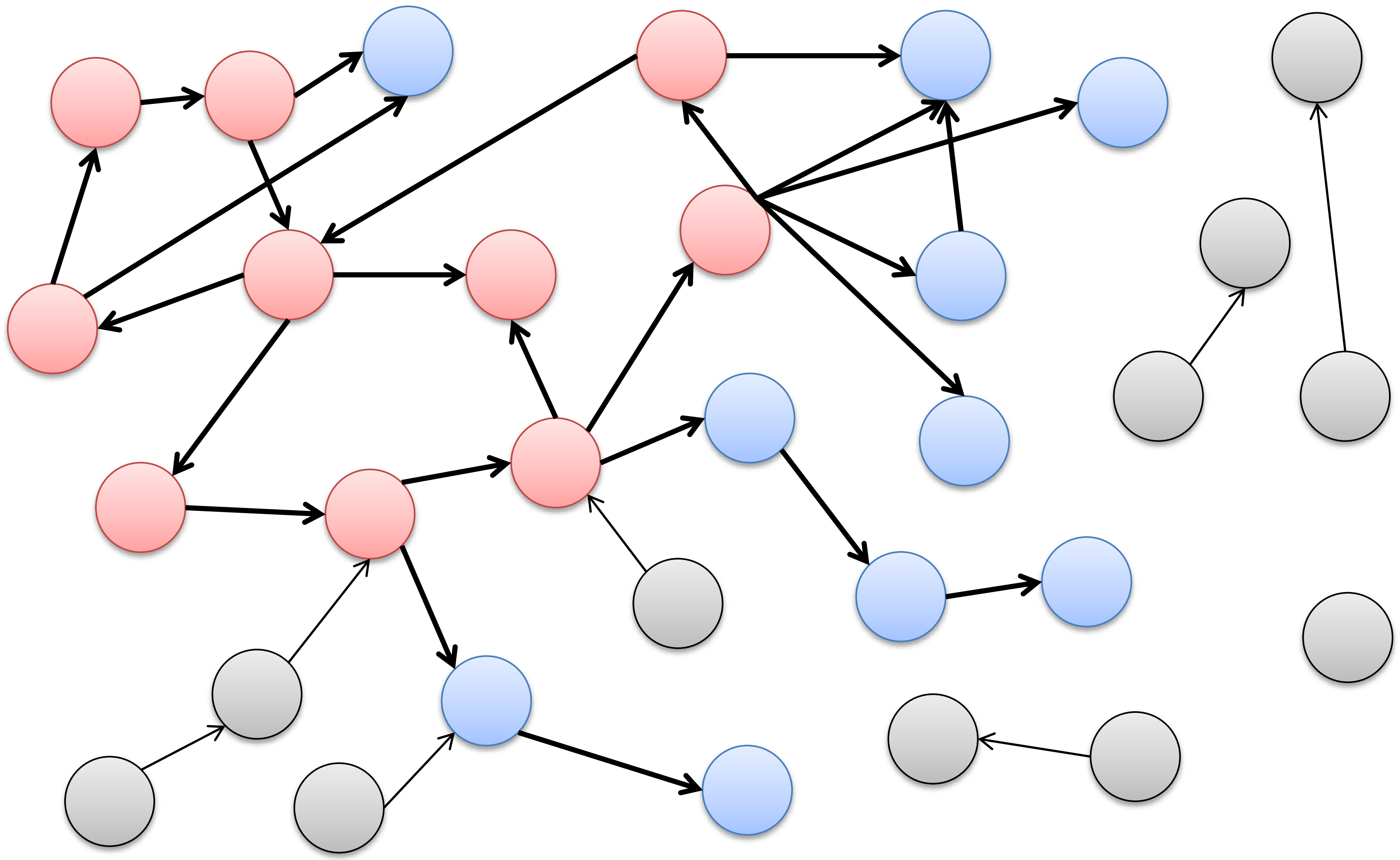
Static Call-Graph



Static Call-Graph



Static Call-Graph



Precision Evaluation

Four operations:

Generic Traversal:	`f => col.foreach{f}`
Pure Traversal:	`col.foreach{ () }`
Impure Traversal:	`col.foreach{ _.visited = true }`
Grow:	`el => col.append(el)`

Four collections:

- immutable.TreeSet
- immutable.List
- mutable.HashSet
- mutable.LinkedList

class Cell(**var** visited = **false**);

Effects Examples

- immutable.TreeSet:

Generic Traversal:	T
Pure Traversal:	Pure
Impure Traversal:	es.tree(.right | .left)*.key.visited
Grow:	Pure

- immutable.List

Generic Traversal:	Pure (conditionally on the closure)
Pure Traversal:	Pure
Impure Traversal:	es.tl*.hd.visited
Grow:	Pure

Effects Examples

- mutable.HashSet

Generic Traversal:	Pure (conditionally on the closure)
Pure Traversal:	Pure
Impure Traversal:	es.table.store.visited
Grow:	es.tableSize | es.table.store | es.sizemap.store | es.sizemap | es.table

- mutable.LinkedList

Generic Traversal:	Pure (conditionally on the closure)
Pure Traversal:	Pure
Impure Traversal:	es.next*.elem.visited
Grow:	es.next.next*

Higher-order Functions

```
def mapHead(l: List, f: Int => Int): List = {
  new List(f(l.head), l.tail)
}
def  test(l: List) {
  mapHead(l, x => x+1)
  mapHead(l, x => {l.tail = null; 0})
}
```

Higher-order Functions

```
def mapHead(l: List, f: Int => Int): List = {
  new List(f(l.head), l.tail)
}
def  test(l: List) {
  mapHead(l, x => x+1)
  mapHead(l, x => {l.tail = null; 0})
}
```

- Reduces to dynamic dispatch:

```
def mapHead(l: List, f: Function1[Int, Int]): List = {
  new List(f.apply(l.head), l.tail)
}
def test(l: List): List = {
   mapHead(l, new Anon1())
   mapHead(l, new Anon2(l))
}
class Anon1 extends Function1[Int, Int] {
   def apply(i: Int) = i + 1
}
class Anon2(l: List) extends Function1[Int, Int] {
   def apply(i: Int) = {l.tail = null; 0 }
}
```

Motivation

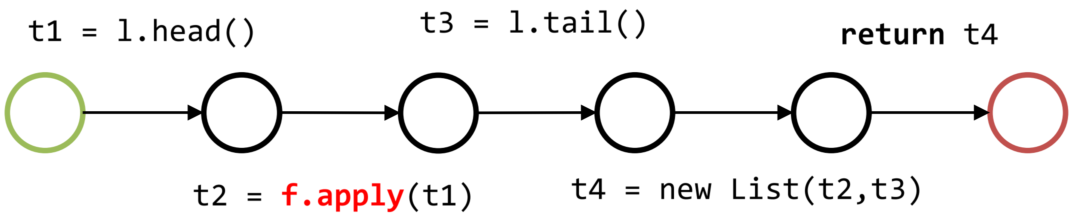
```
class Cell {
   var visited = false
}

def toggle(c: Cell) {
   c.visited = !c.visited
}

def apply(c: Cell, f: Function1[Cell, Unit]) {
   f.apply(c)
}

def visitAll(cs: List[Cell]) {
   cs.foreach(new Closure1())
}

class Closure1() extends Function1[Cell, Unit] {
   def apply(c: Cell) { c.visited = true }
}
```

Delaying Effect Composition

```
def mapHead(l: List, f: Function1[Int, Int]): List = {
  new List(f.apply(l.head), l.tail)
}
```

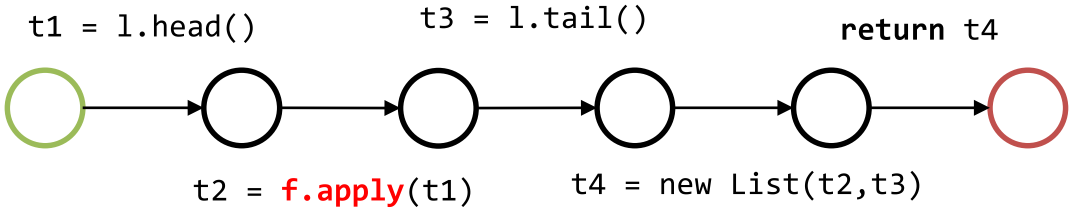
Delaying Effect Composition

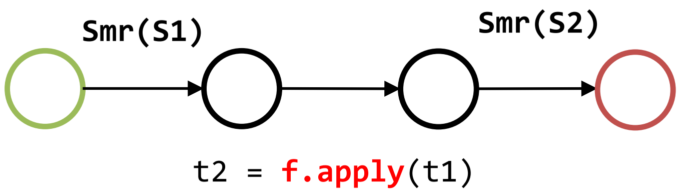
```
def mapHead(l: List, f: Function1[Int, Int]): List = {
  new List(f.apply(l.head), l.tail)
}
```



Delaying Effect Composition

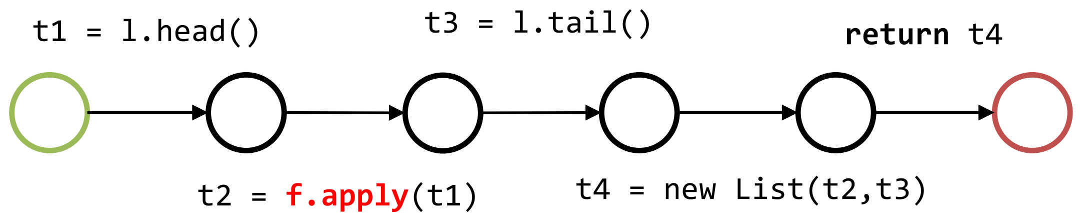
```
def mapHead(l: List, f: Function1[Int, Int]): List = {
  new List(f.apply(l.head), l.tail)
}
```

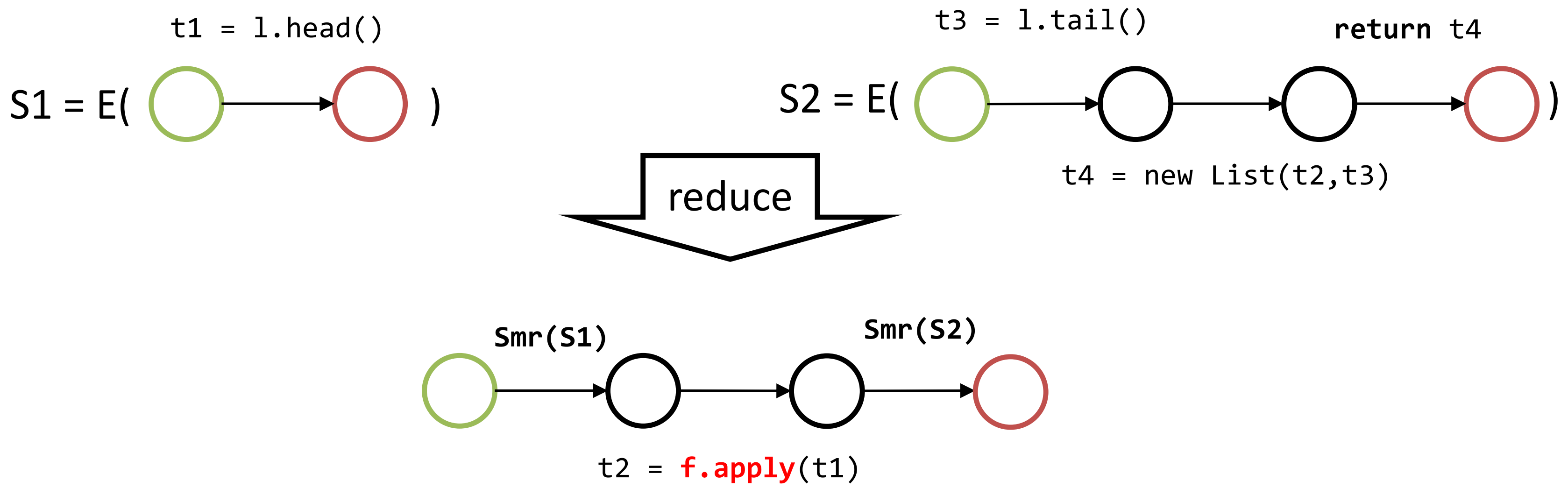




Delaying Effect Composition

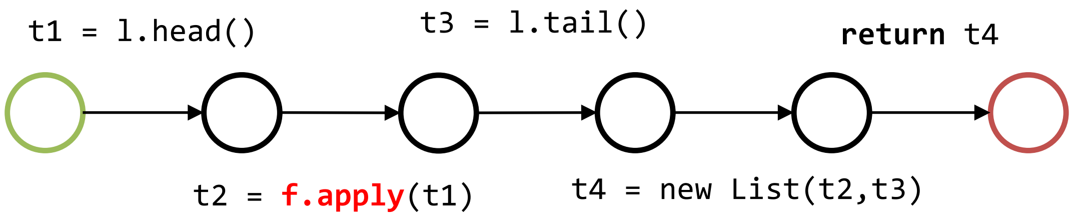
```
def mapHead(l: List, f: Function1[Int, Int]): List = {
  new List(f.apply(l.head), l.tail)
}
```

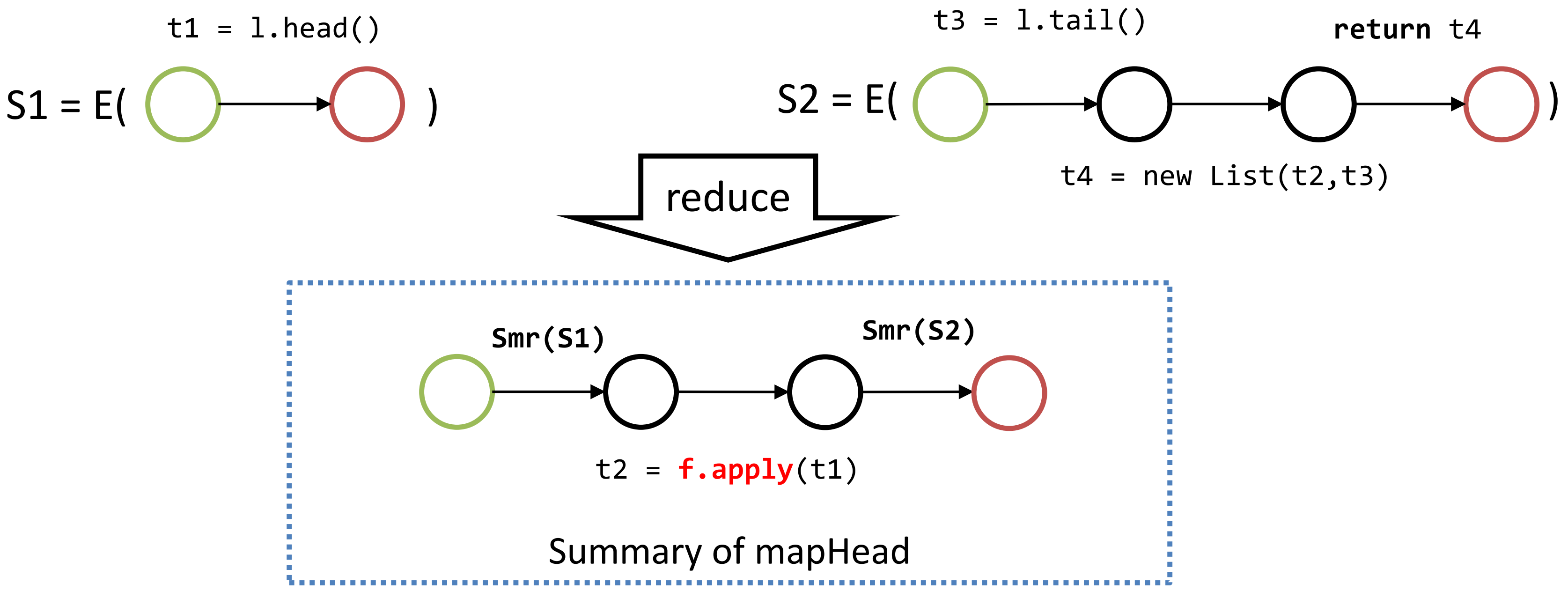




Delaying Effect Composition

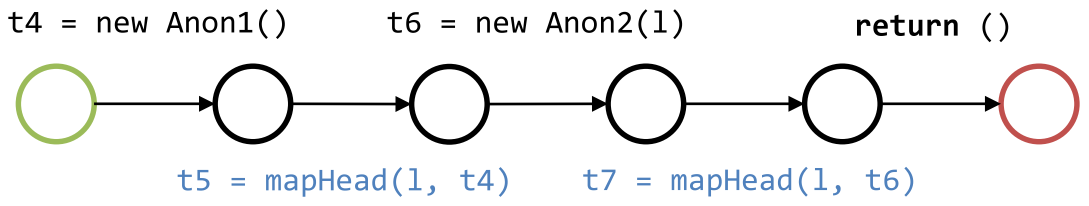
```
def mapHead(l: List, f: Function1[Int, Int]): List = {
  new List(f.apply(l.head), l.tail)
}
```





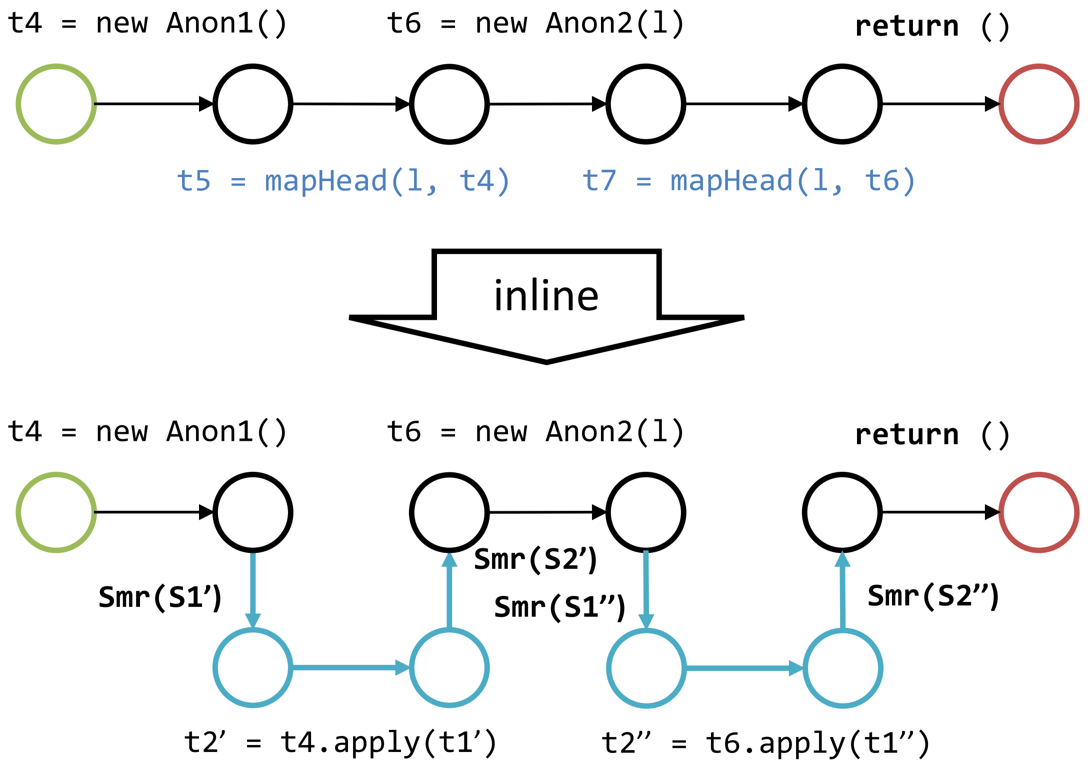
Delaying Effect Composition

```
def test(l: List) {
   mapHead(l, new Anon1())
   mapHead(l, new Anon2(l))
}
```



Delaying Effect Composition

```
def test(l: List) {
   mapHead(l, new Anon1())
   mapHead(l, new Anon2(l))
}
```



Results

Package	#Methods	Pure	Cond. Pure	Impure	T
scala	5721	79%	11%	10%	1%
scala.annotation	41	93%	2%	2%	2%
scala.beans	25	64%	8%	29%	8%
scala.collection	34810	46%	17%	29%	8%
scala.compat	9	22%	33%	44%	0%
scala.io	546	47%	11%	40%	2%
scala.math	1847	67%	28%	5%	0%
scala.parallel	39	77%	23%	0%	0%
scala.runtime	113	58%	3%	39%	0%
scala.sys	5862	50%	9%	40%	1%
scala.testing	44	52%	1%	43%	2%
scala.text	115	87%	0%	11%	2%
scala.util	1786	51%	11%	32%	6%
scala.util.parsing	2206	56%	12%	27%	5%
scala.xml	2860	56%	11%	30%	3%
Total	**58410**	**52%**	**15%**	**27%**	**6%**

Composition

- We define composition as applying an effect within another effect:

$$E_{res} = E_{outer} \Diamond E_{inner}$$

satisfying:

$$\gamma(E_{res}) \supseteq \gamma(E_{outer}) \circ \gamma(E_{inner})$$

$$\gamma: E \to 2^{H \times H}$$