Topics related to the Expression Problem

including but not limited to

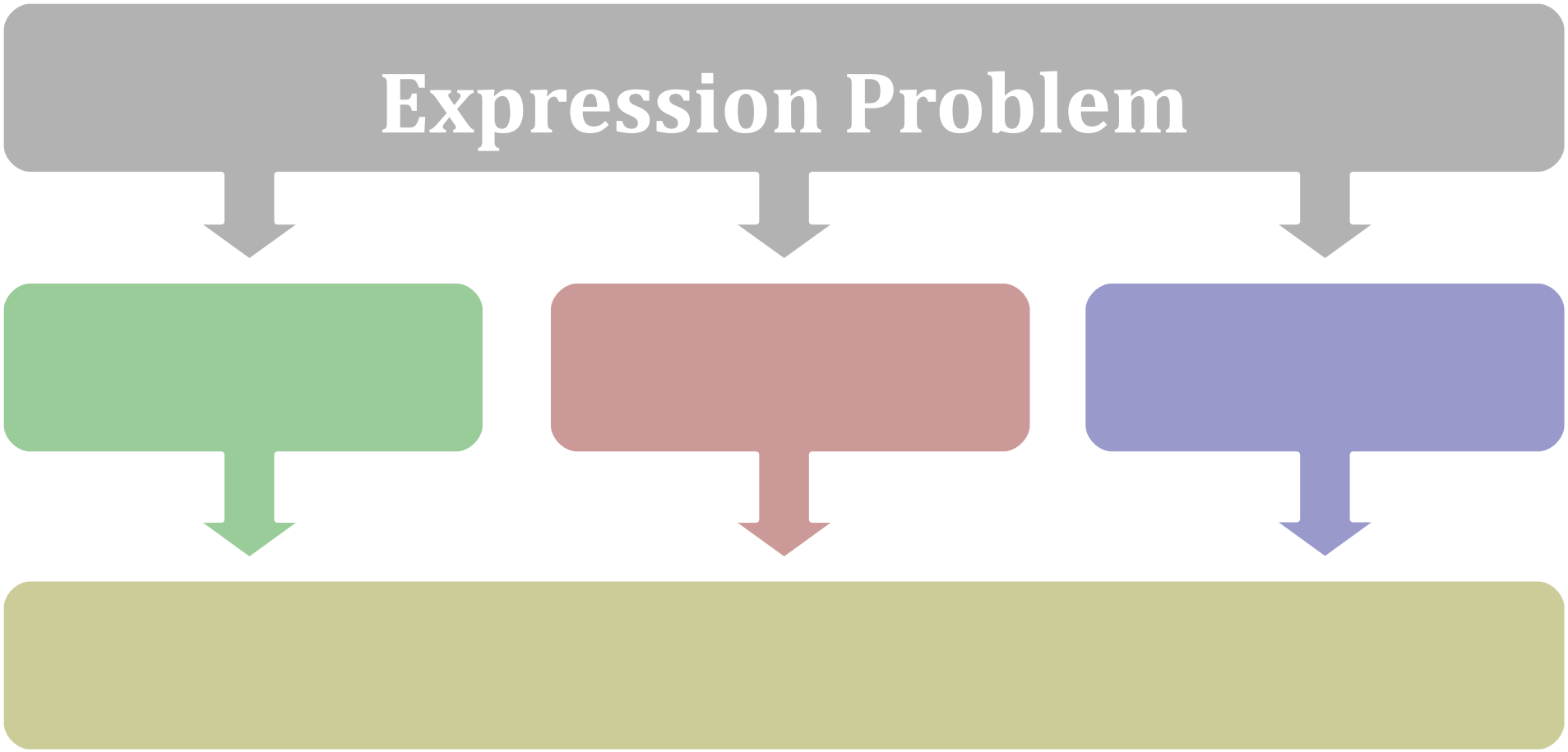
Compositional and Linear Encoding (of Synthesized and Inherited Attributes as Object Algebras in Scala)

Tillmann Rendel

University of Tübingen, Germany

Invited Talk by Tillmann Rendel at the
Workshop on Generic Programming,
Vancouver, British Columbia, August 30, 2015

Overview Map



Expression Problem

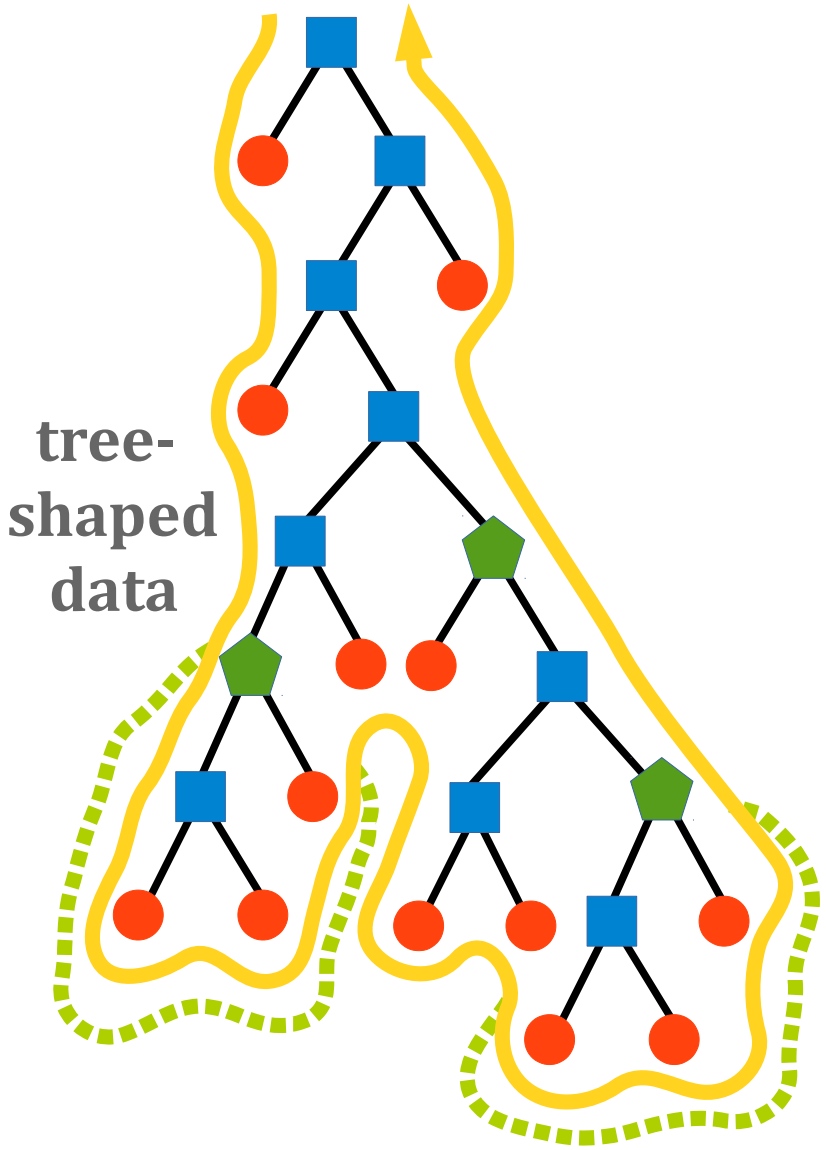
Wadler 1998 (The Expression Problem)

Expression Problem

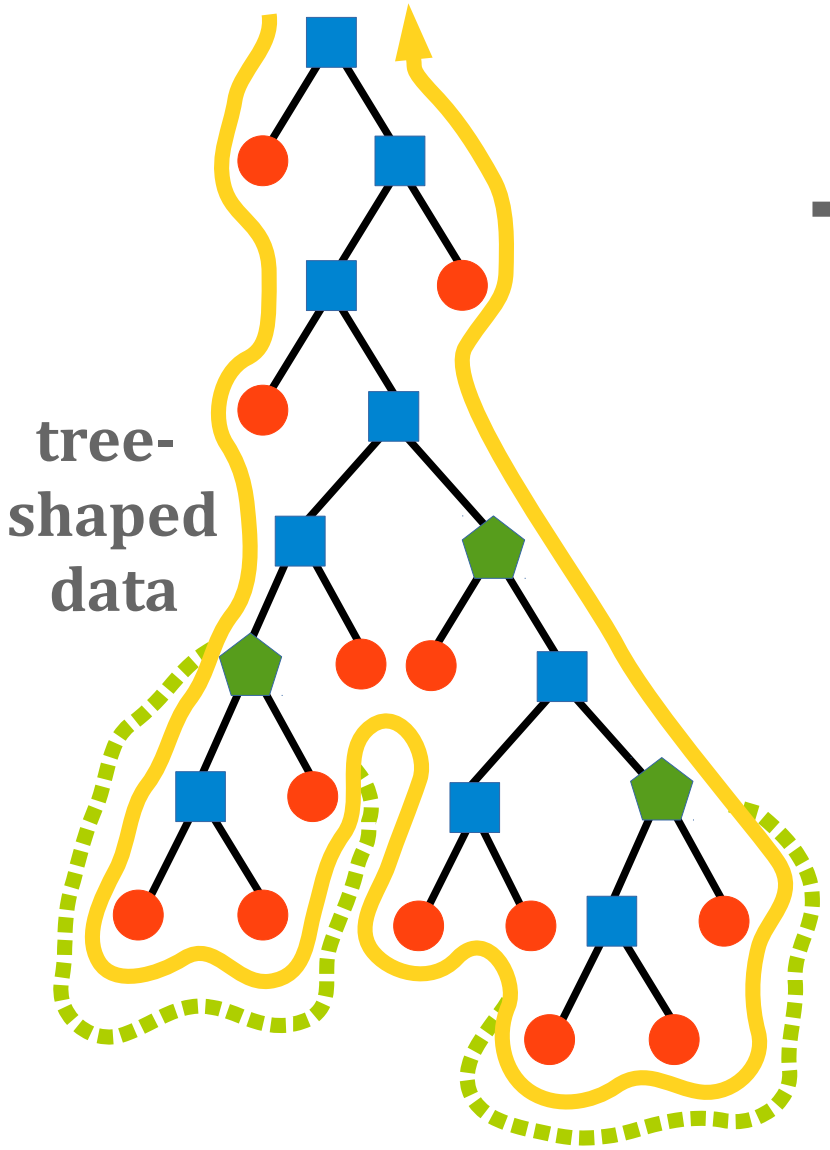
tree-shaped data

Expression Problem

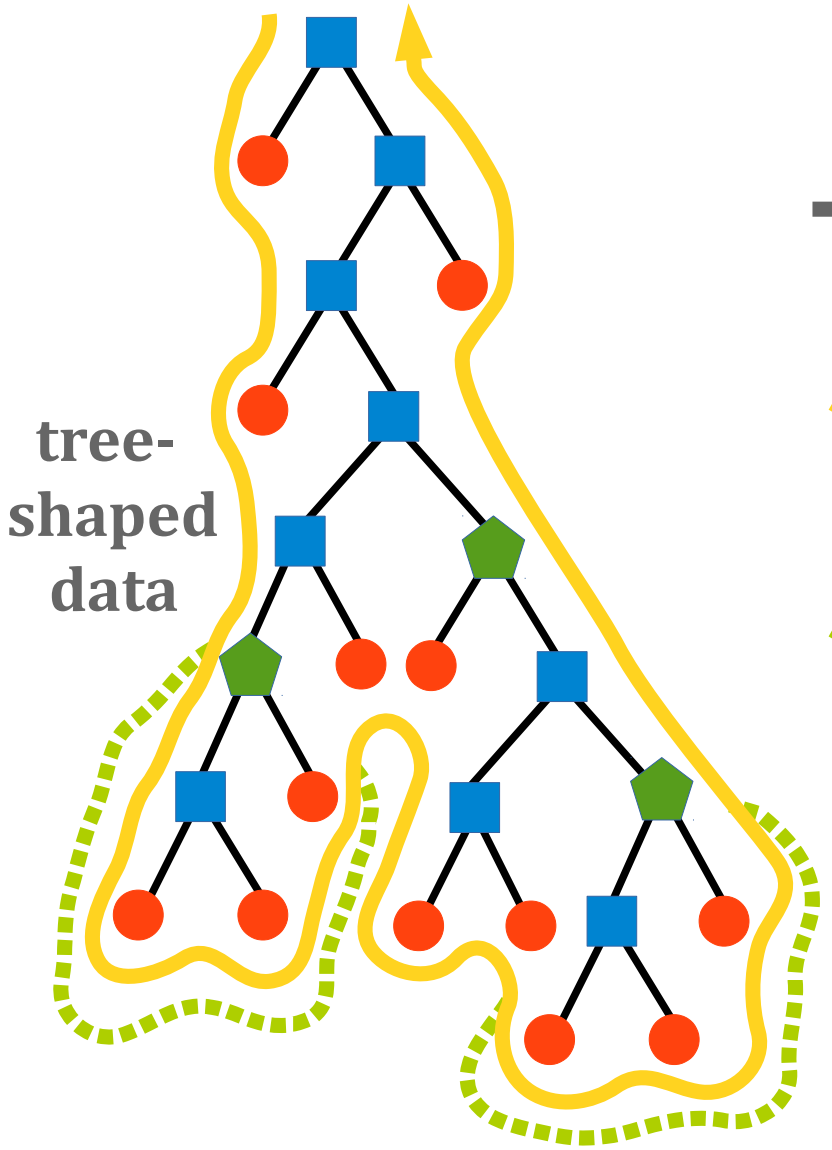
Expression Problem

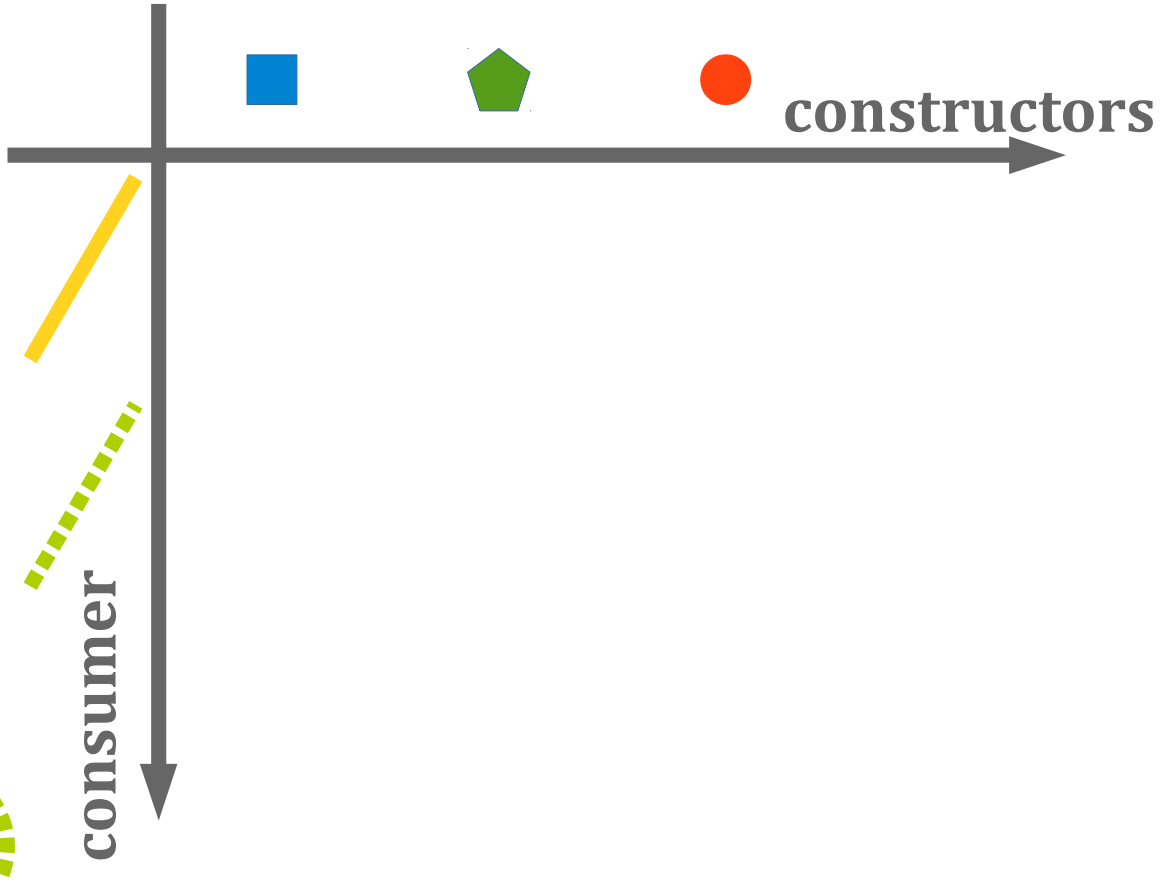


Expression Problem

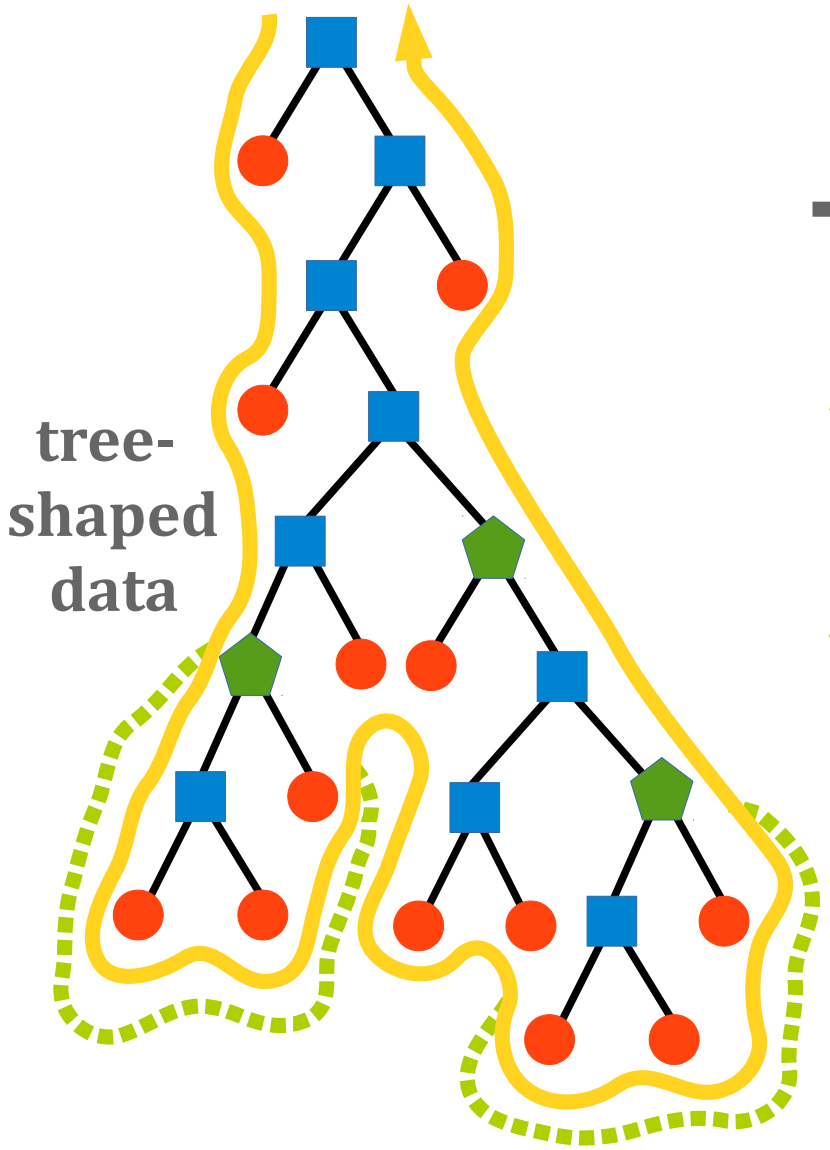


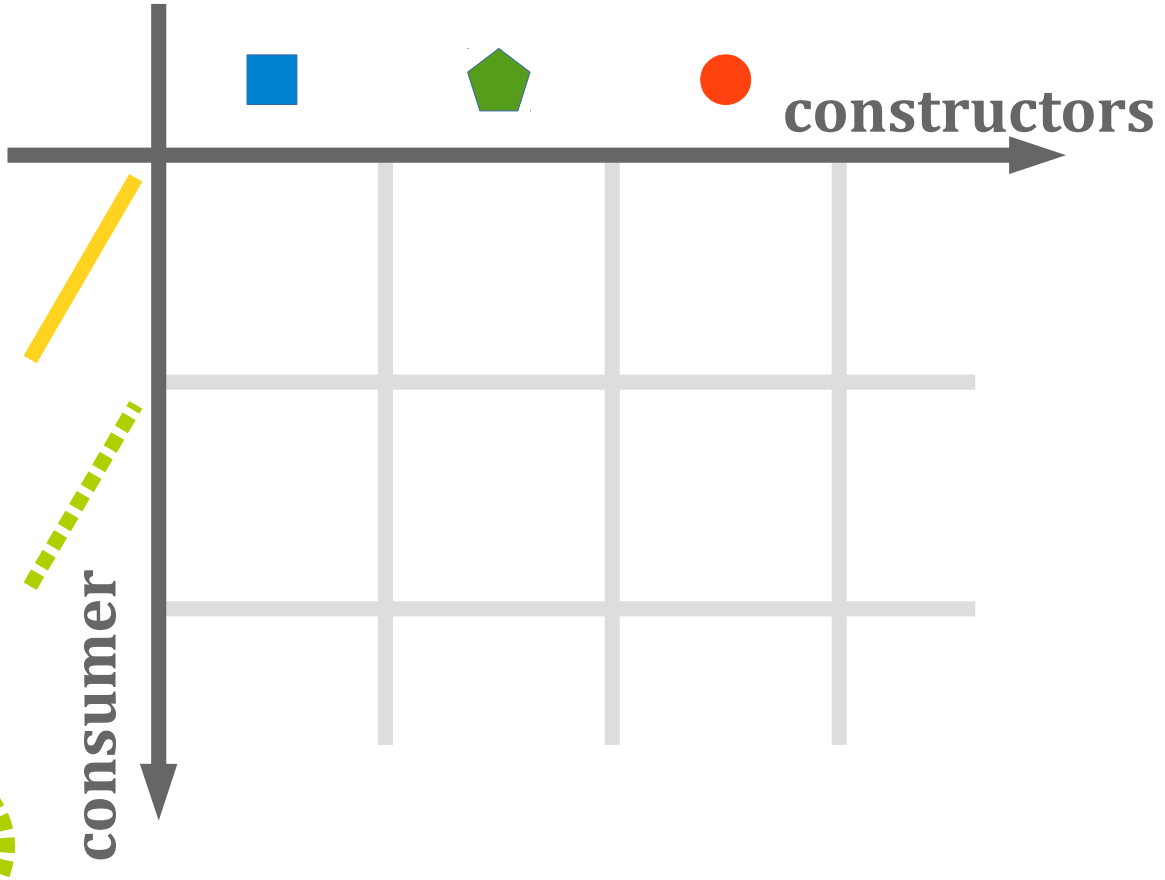
Expression Problem



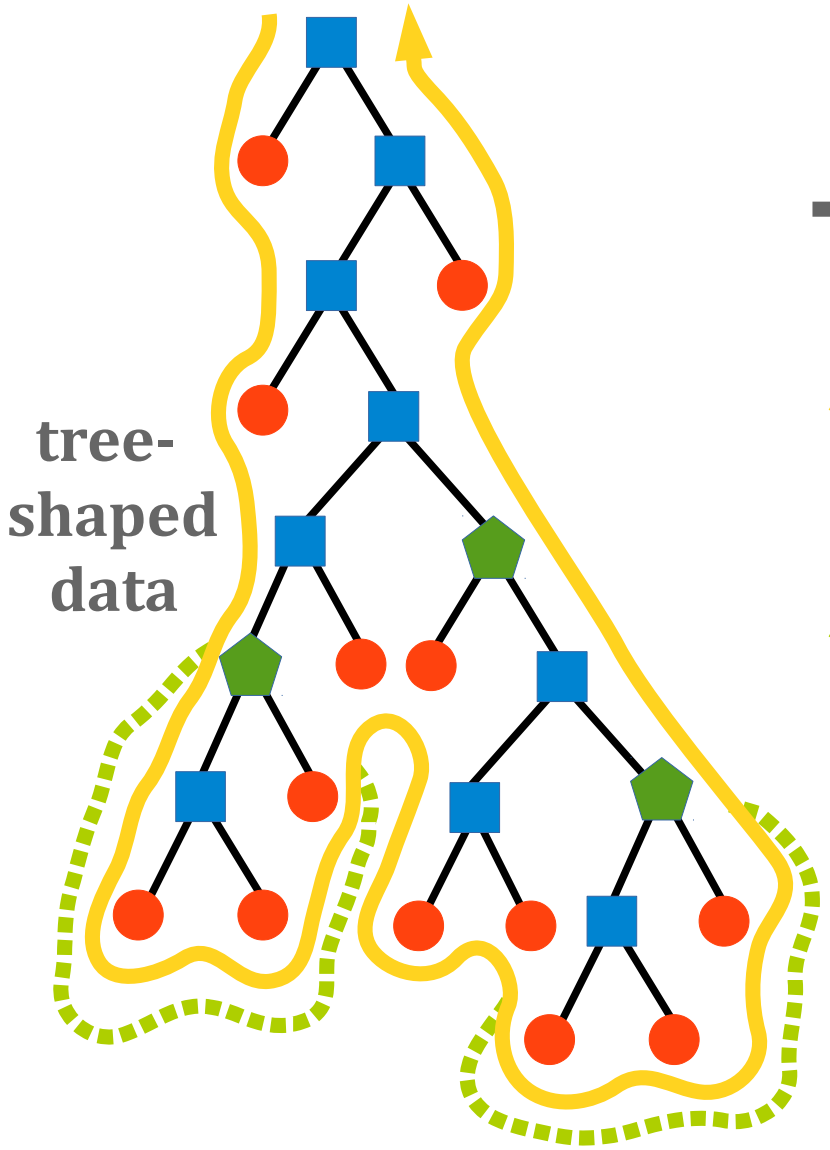


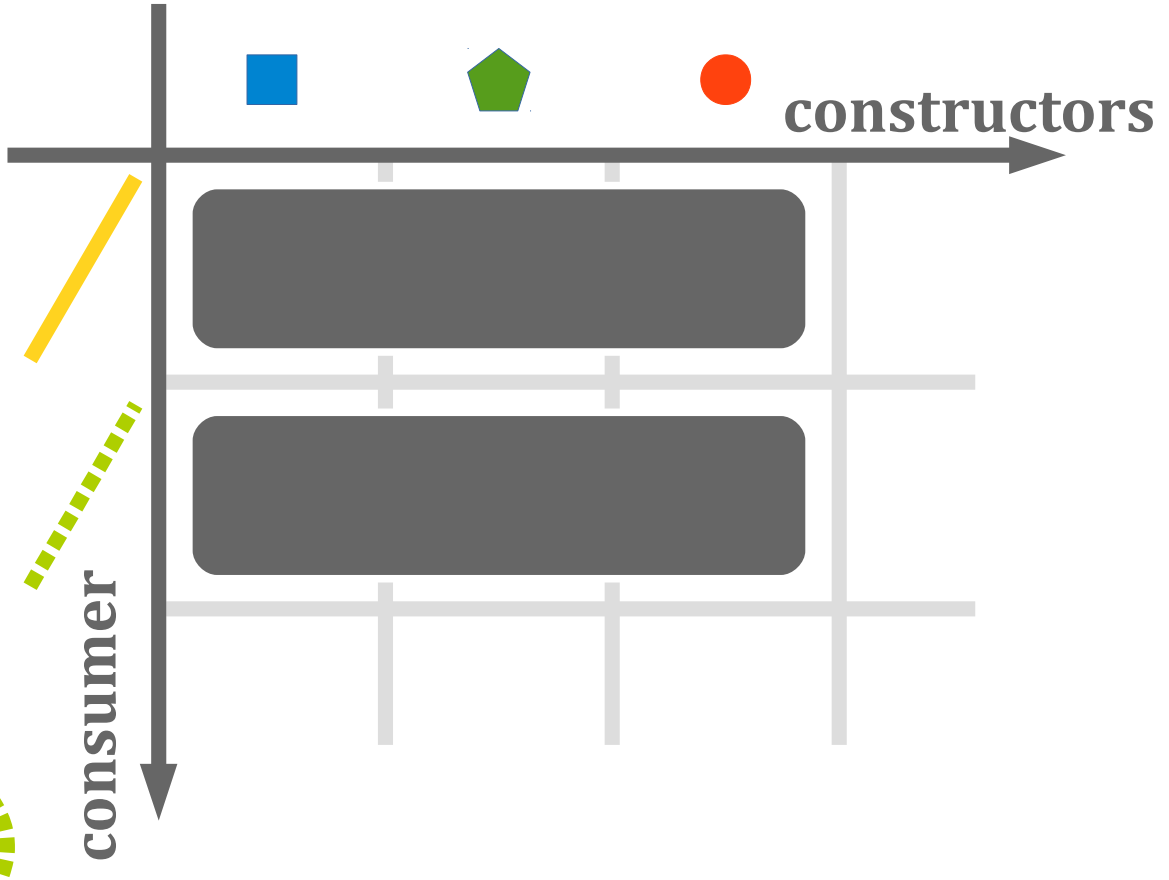
Expression Problem



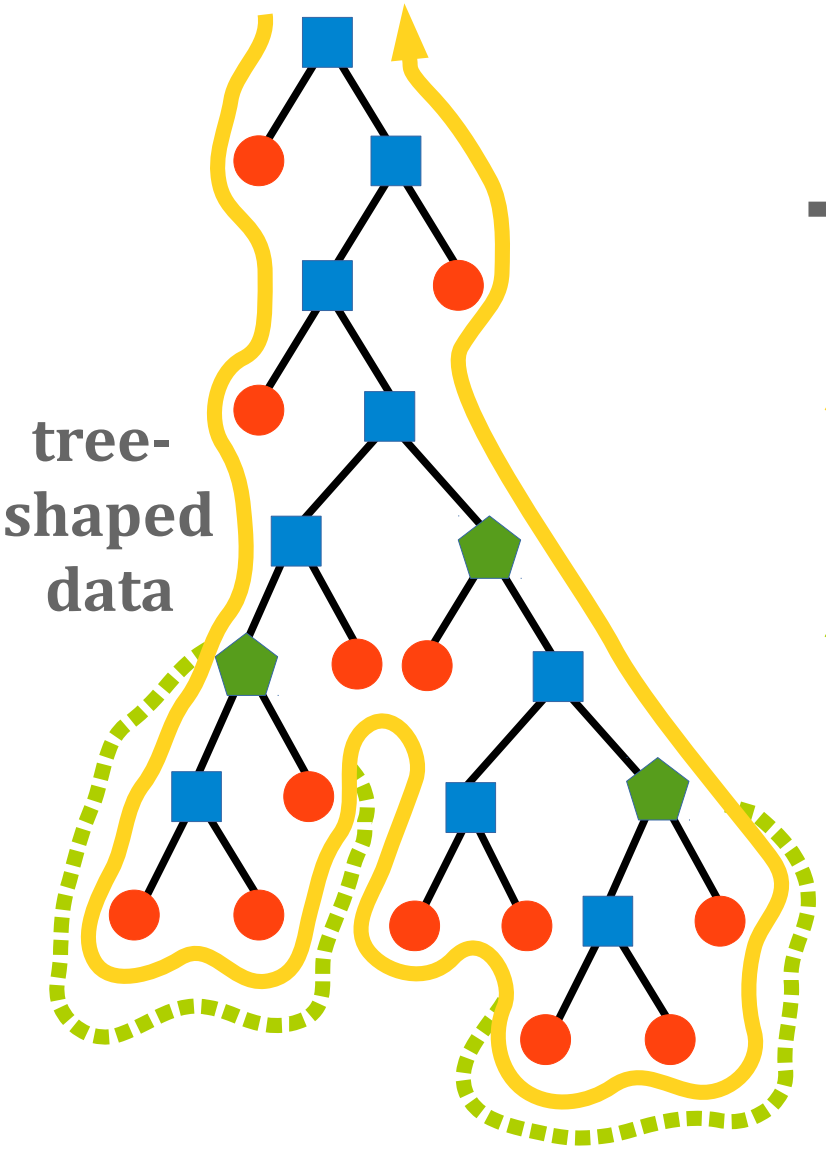


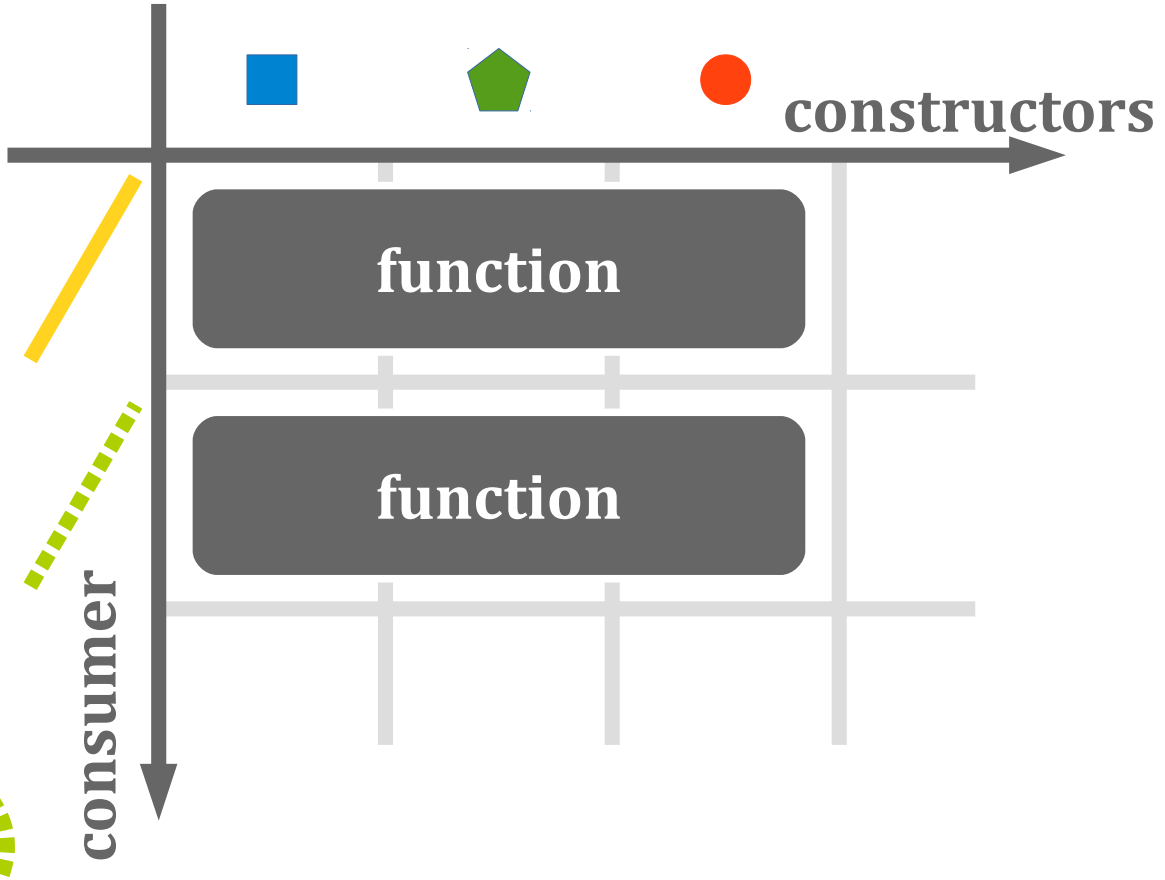
Expression Problem



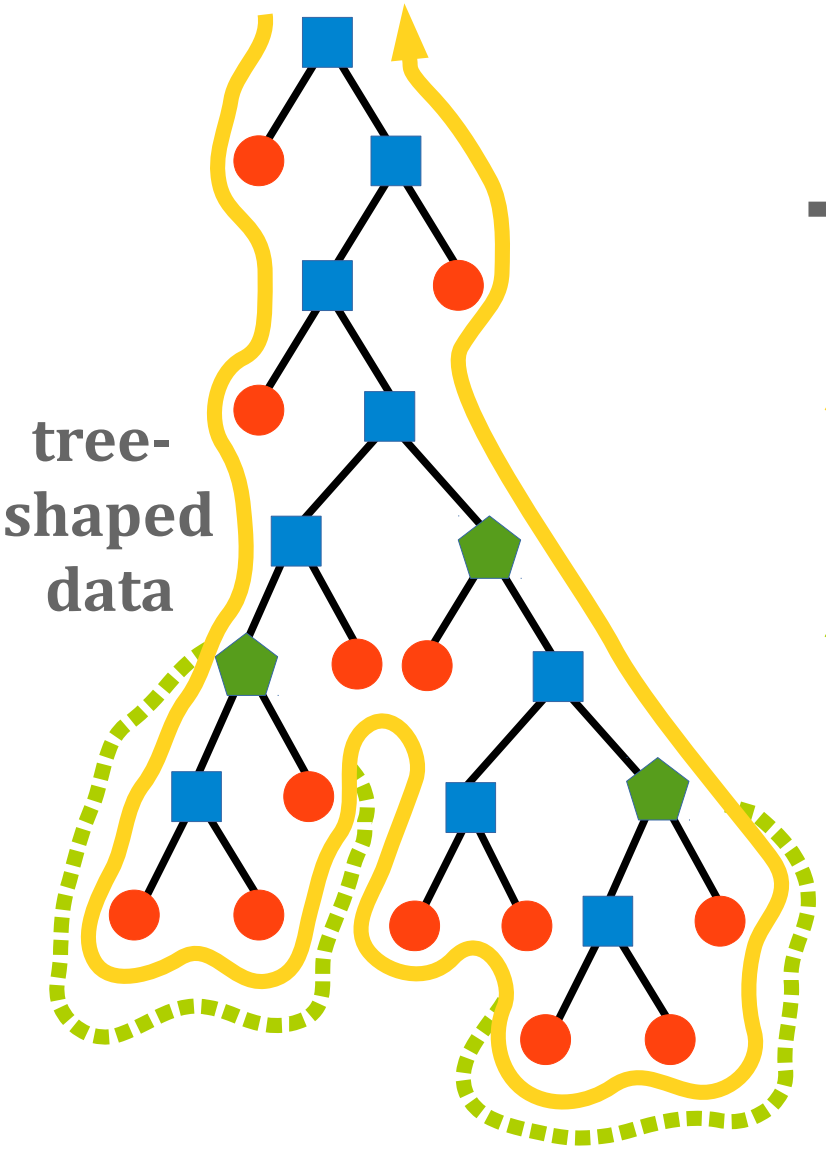


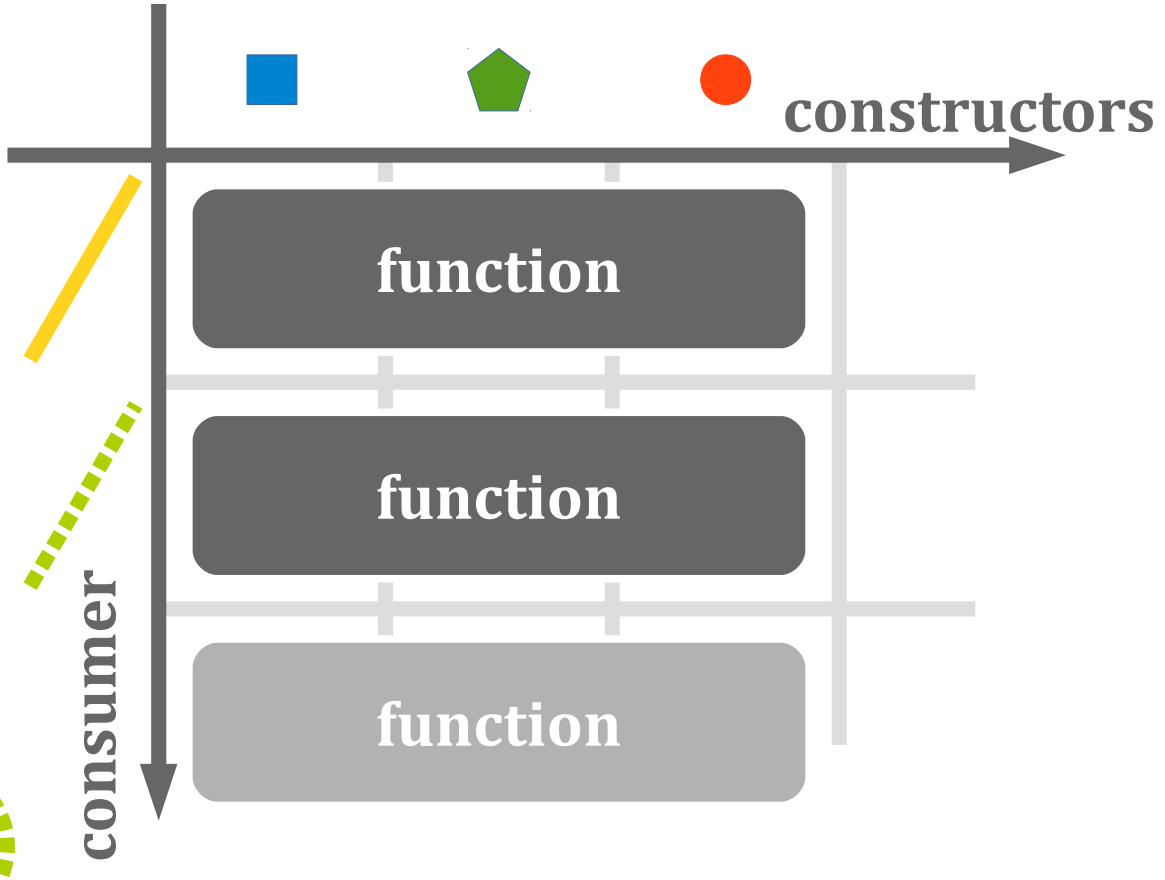
Expression Problem



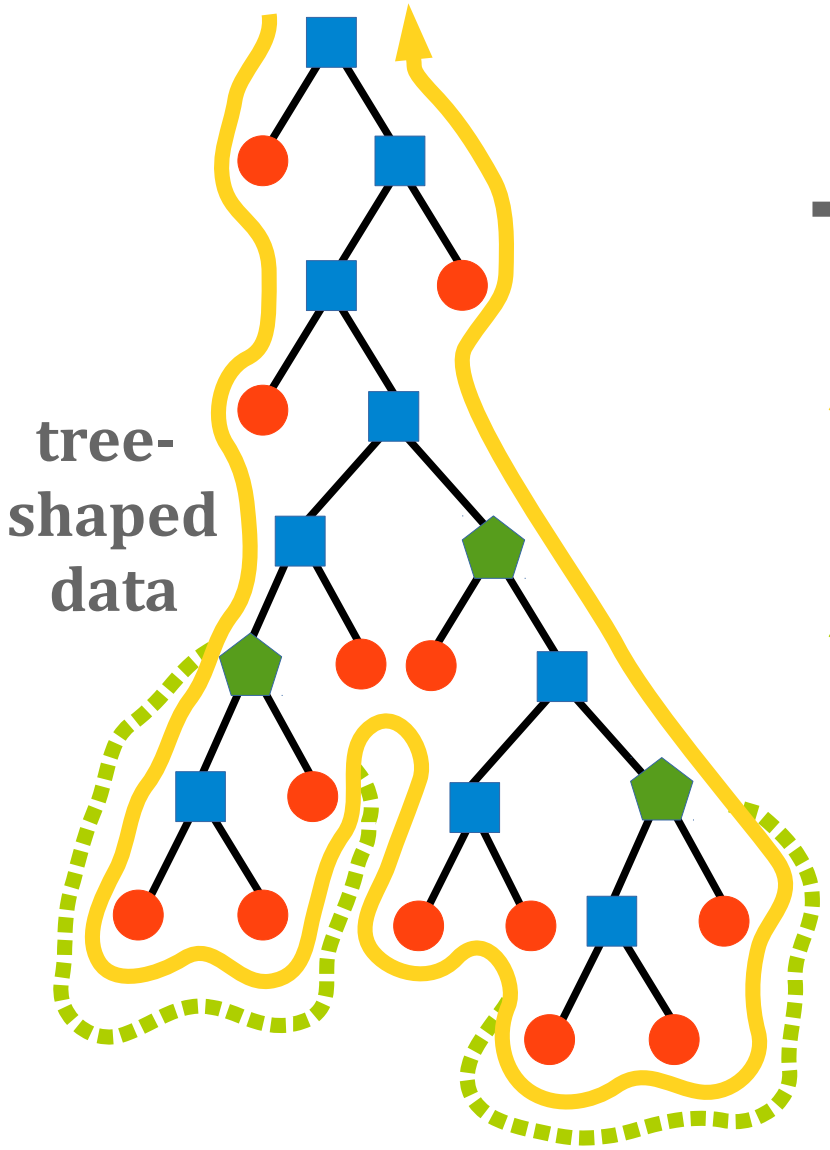


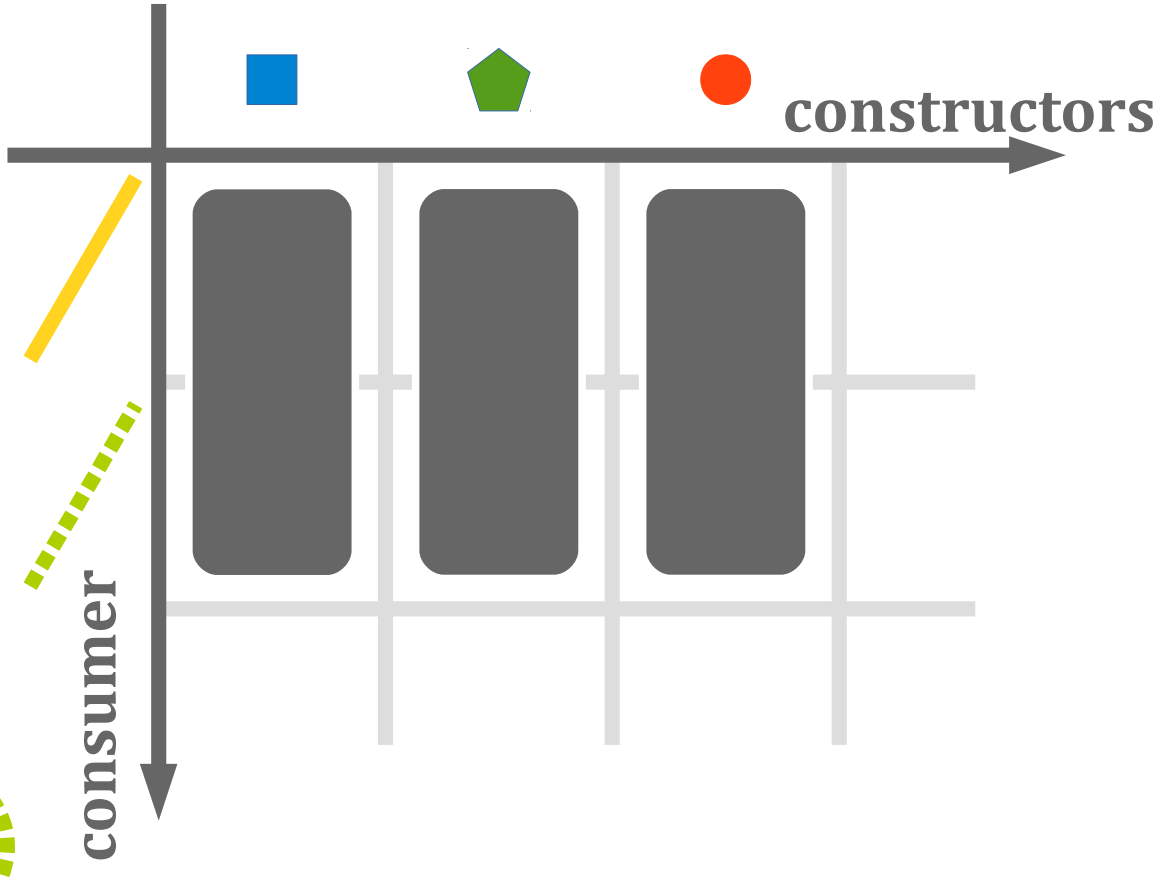
Expression Problem



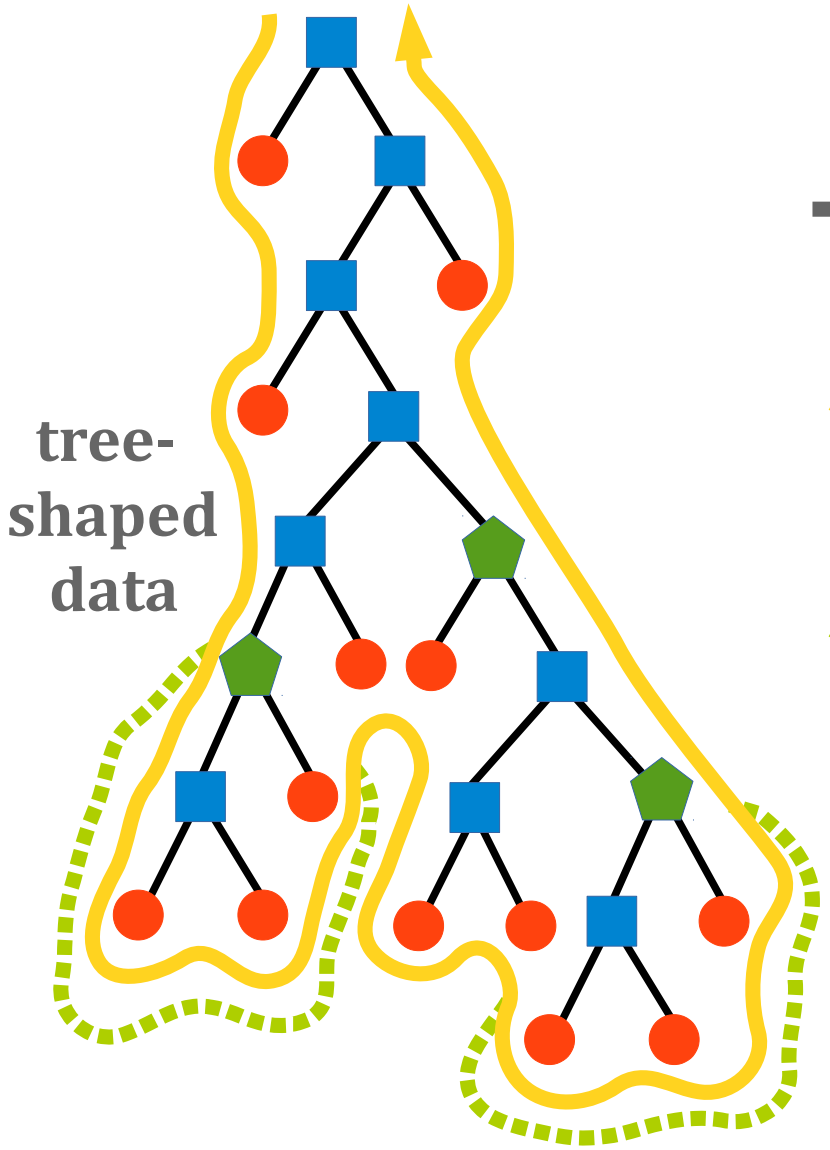


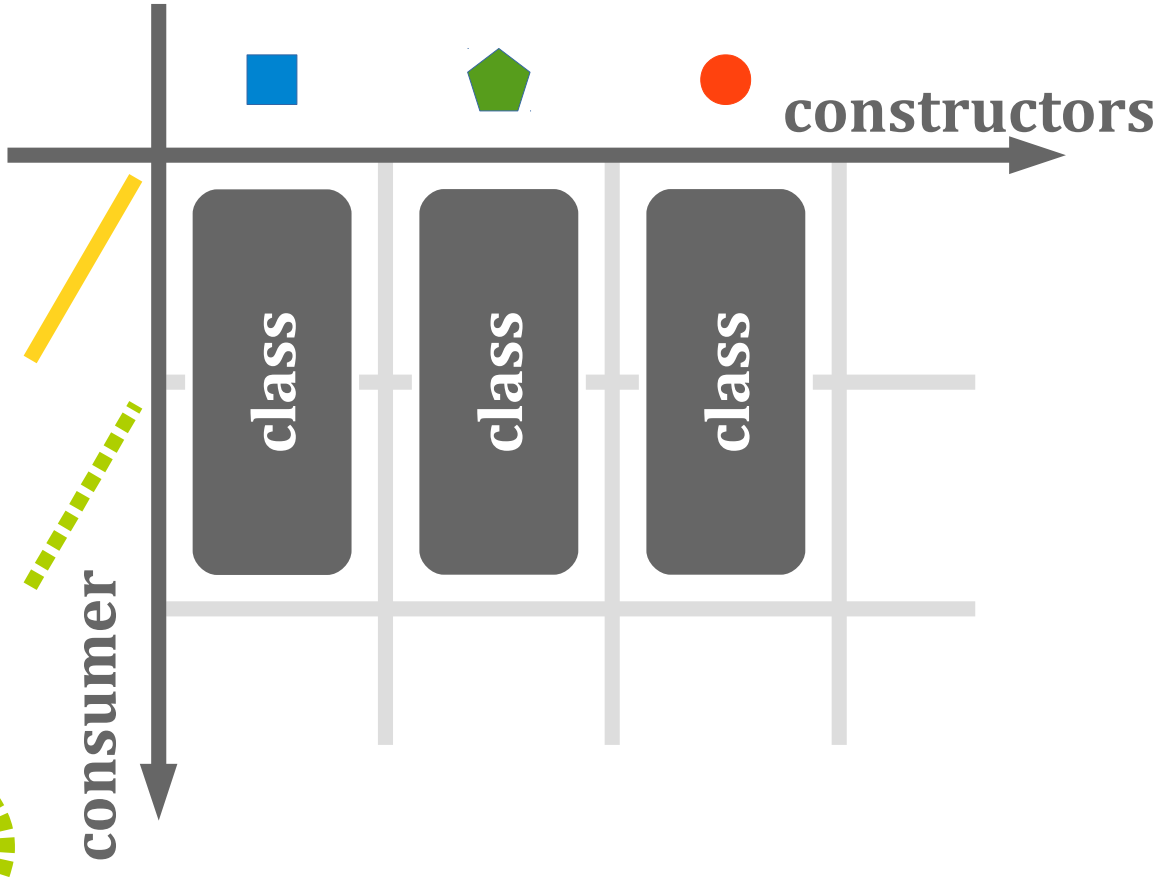
Expression Problem



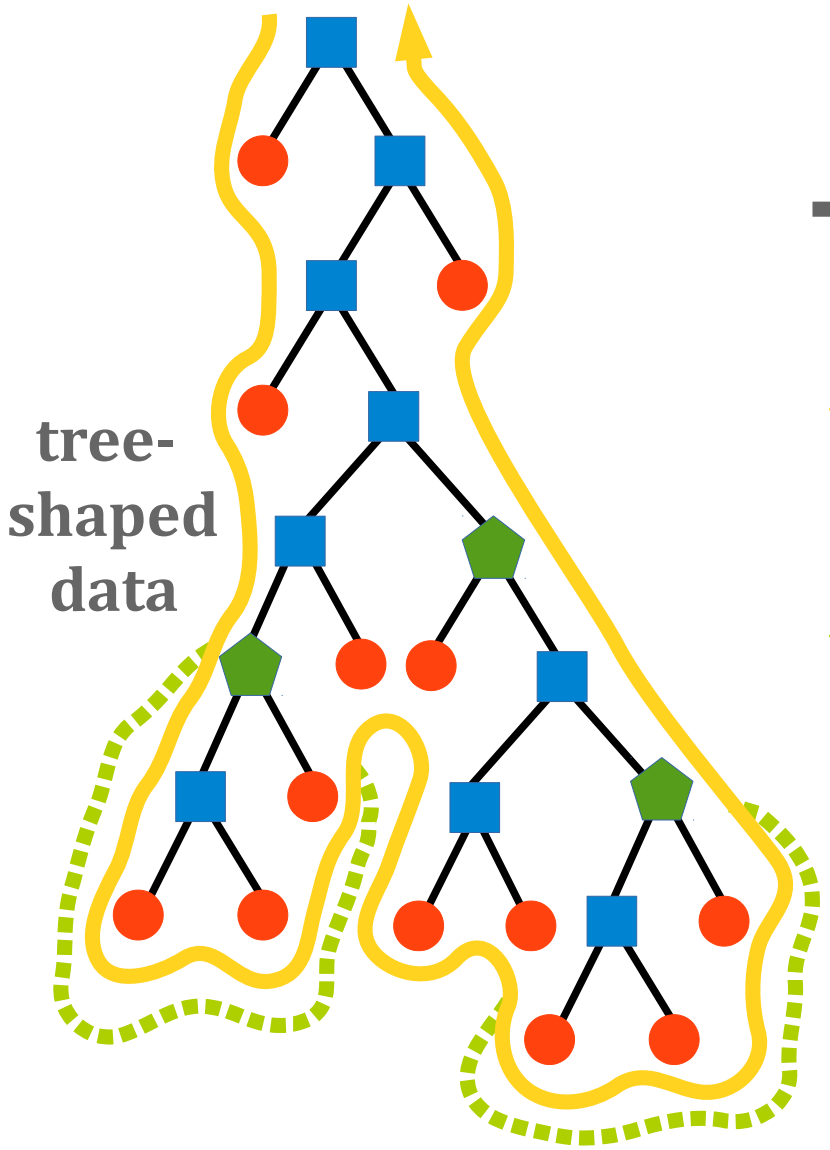


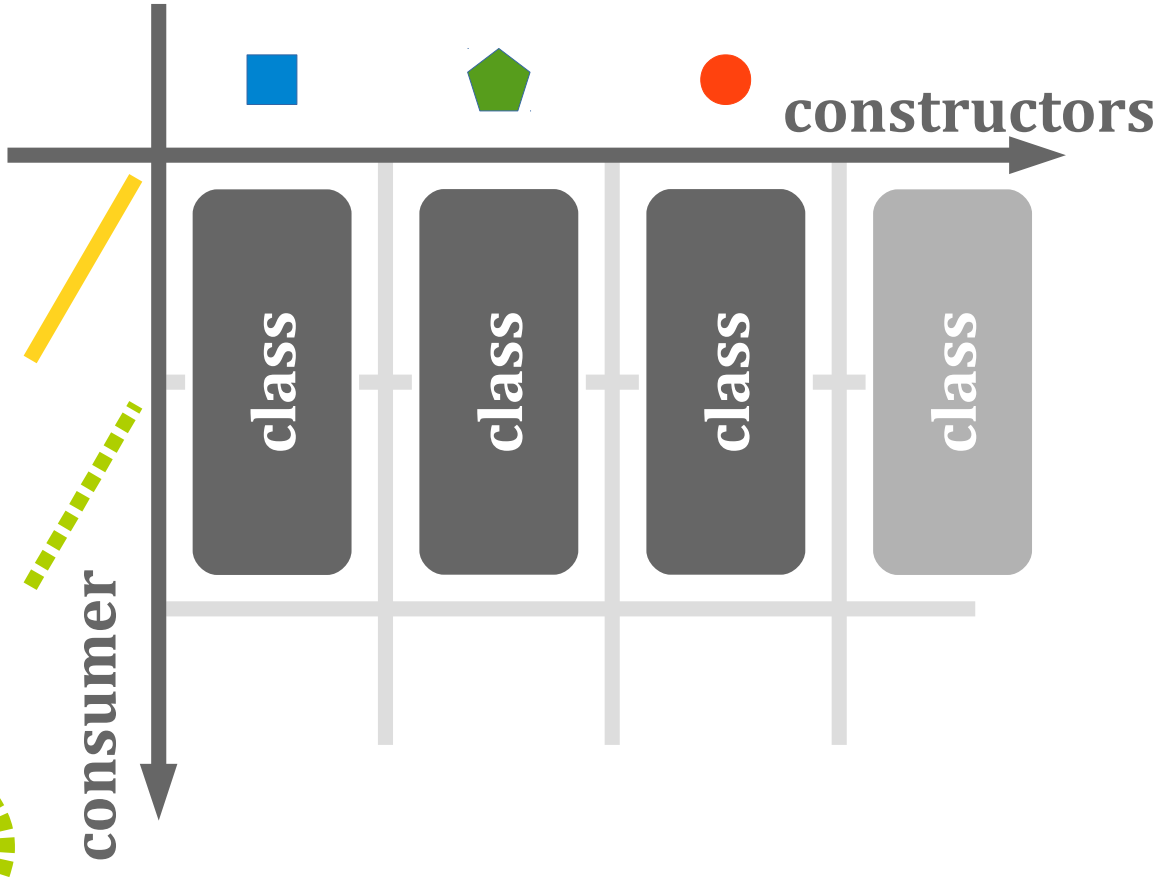
Expression Problem



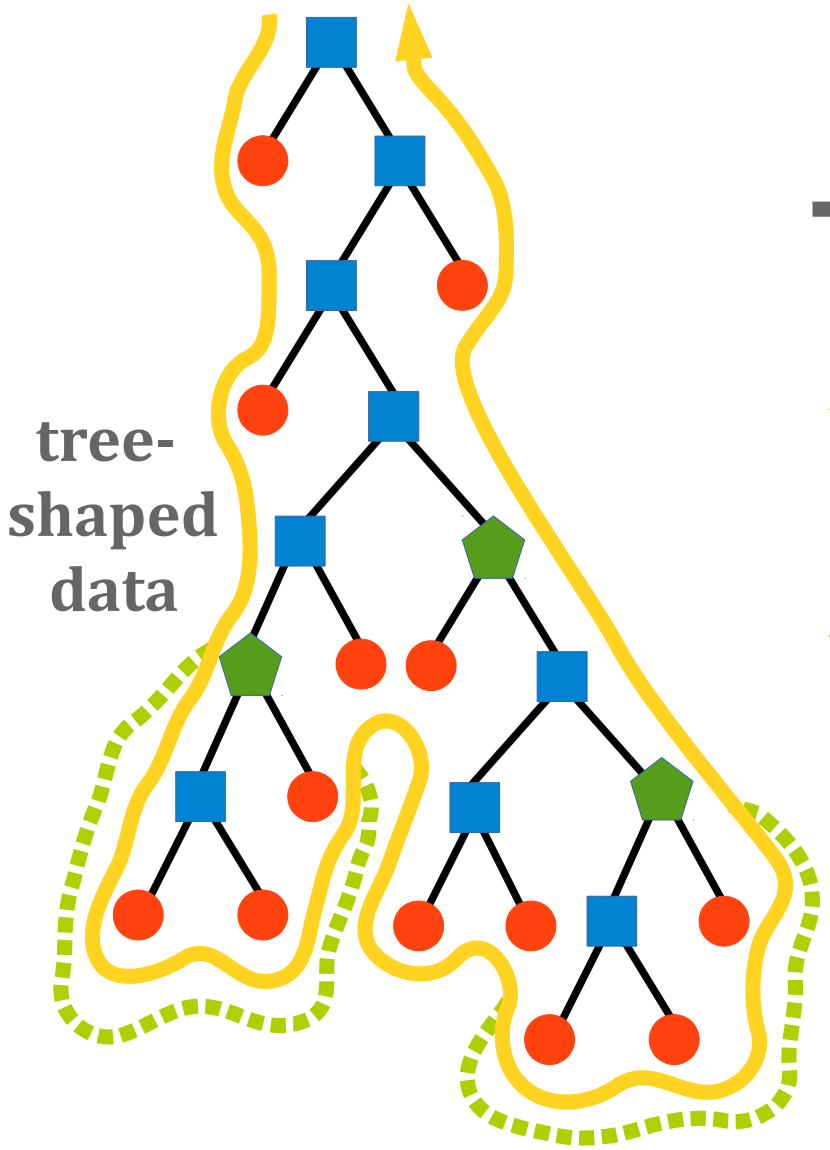


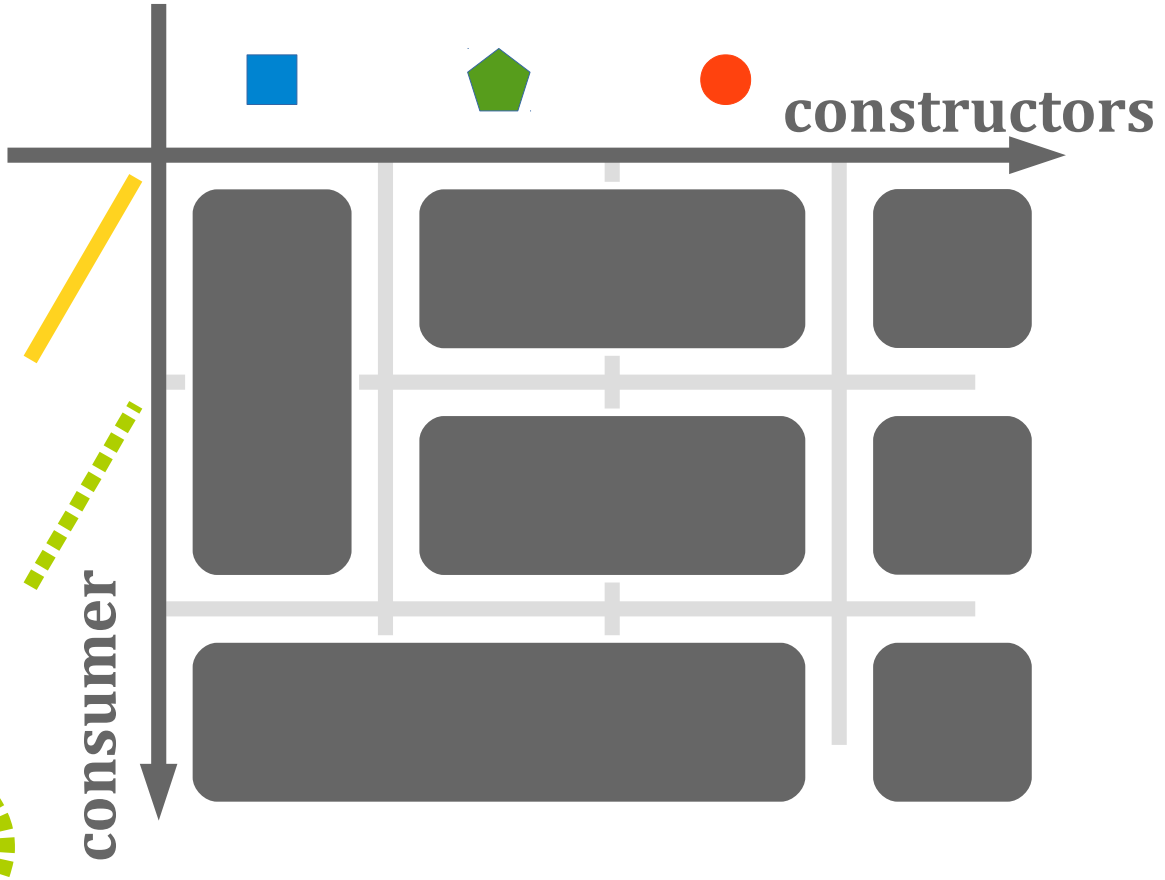
Expression Problem



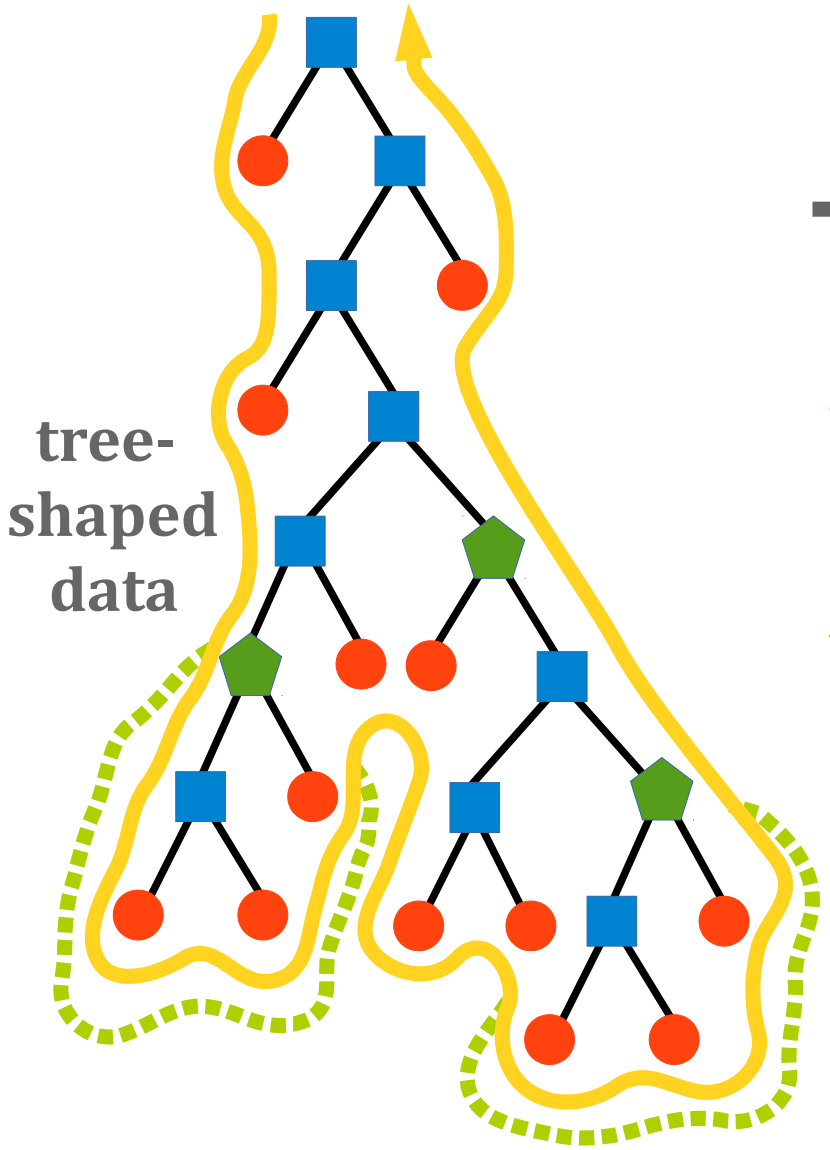


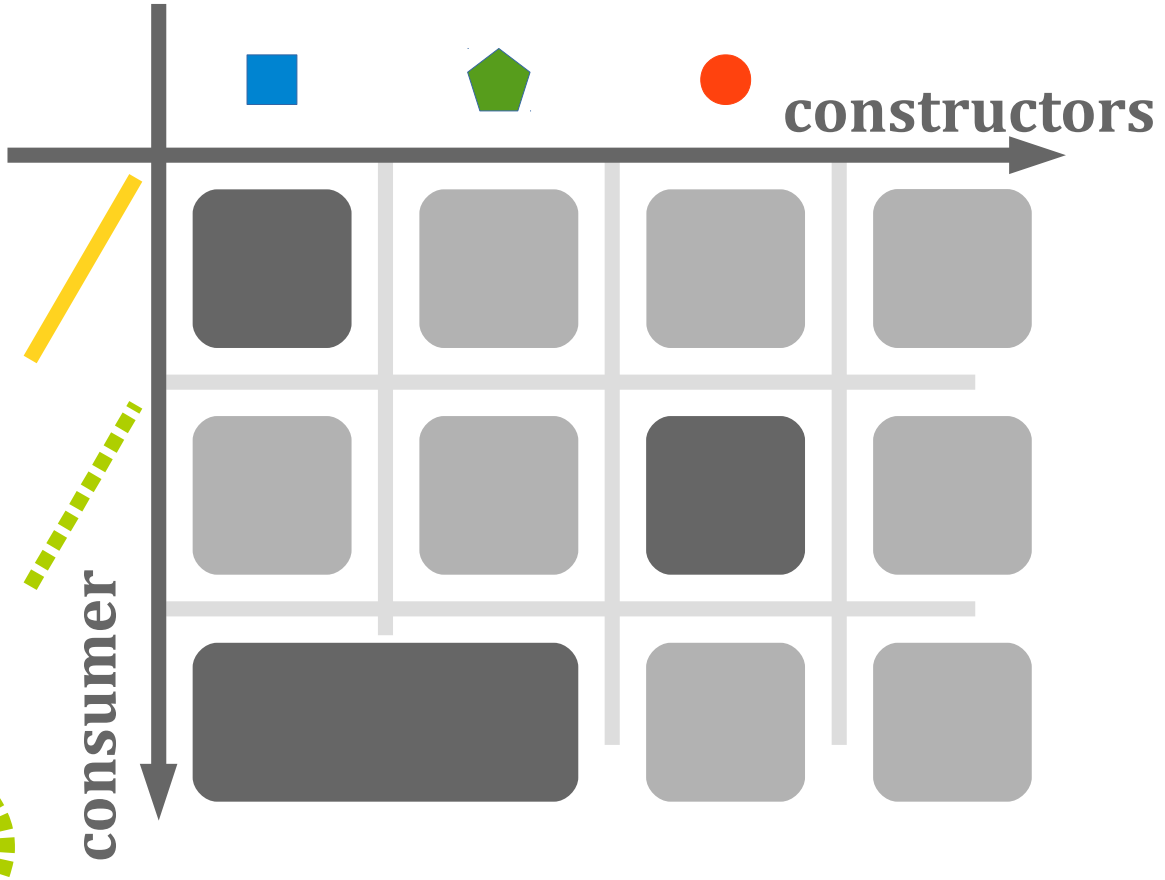
Expression Problem

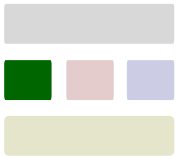




Expression Problem

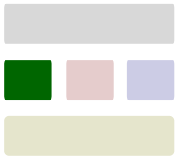






Deep, Shallow and Final* Embedding

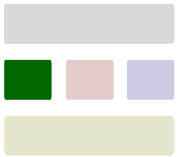
*also known as „polymorphic embedding“ or „object algebras“



Deep, Shallow and Final* Embedding

*also known as „polymorphic embedding“ or „object algebras“

Carette, Kiselyov, Shan 2007, 2009 (Finally Tagless Emb. ...)
Hofer, Rendel, Ostermann, De Moors 2008 (Polymorphic Emb. ...)
Oliveira, Cook 2012 (Extensibility for the Masses: ...)

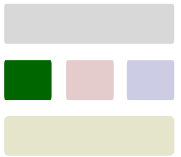


Extensible Records

Some languages with basic or no support for extensible algebraic data types still support extensible and very expressive record types.

- Haskell: type classes, multiple constraints
- Scala: traits, mixin composition
- Java: interfaces, multiple upper bounds

Many solutions to the expression problem represent variants as records to benefit from advanced record support.



Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
   case Lit(n) => n
   case Add(l, r) => eval(l) + eval(r)
 }
```

Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp
```

```
def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```

Deep Embedding

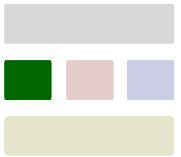
```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```

Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```

Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```



Deep Embedding

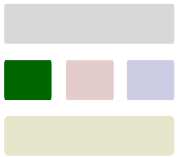
```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```

Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```

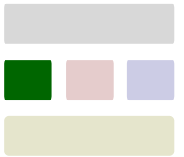
Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```

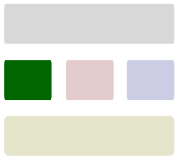


Deep Embedding

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
   case Lit(n) => n
   case Add(l, r) => eval(l) + eval(r)
 }
```



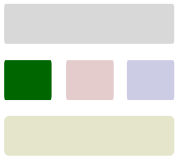
```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
}
```



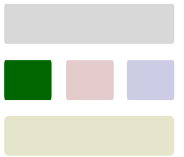
```
trait Exp { def eval: Int }
```

```
case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp
```

```
  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
```

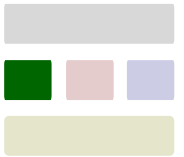


```
trait Exp { def eval: Int }
case class Lit(n: Int) extends Exp
case class Add(l: Exp, r: Exp) extends Exp

  case Lit(n) => n
  case Add(l, r) => eval(l) + eval(r)
```



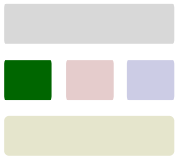
```
trait Exp { def eval: Int }
def Lit(n: Int) = new Exp { def eval = n }
case class Add(l: Exp, r: Exp) extends Exp
```

```
case Add(l, r) => eval(l) + eval(r)
```

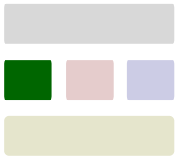


```
trait Exp { def eval: Int }

def Lit(n: Int) = new Exp { def eval = n }

case class Add(l: Exp, r: Exp) extends Exp
```

```
case Add(l, r) => eval(l) + eval(r)
```



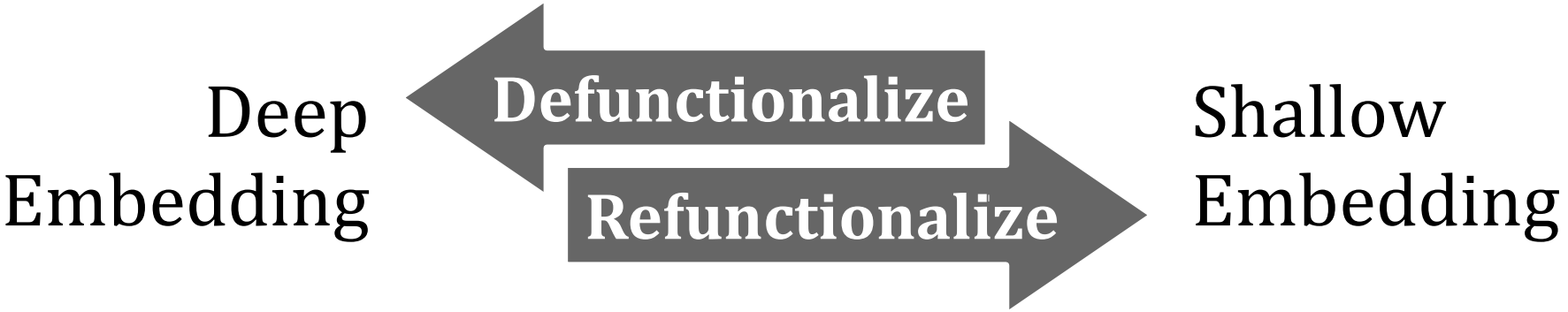
```
trait Exp { def eval: Int }

def Lit(n: Int) = new Exp { def eval = n }

def Add(l: Exp, r: Exp) =
  new Exp { def eval = l.eval + r.eval }
```



Shallow Embedding

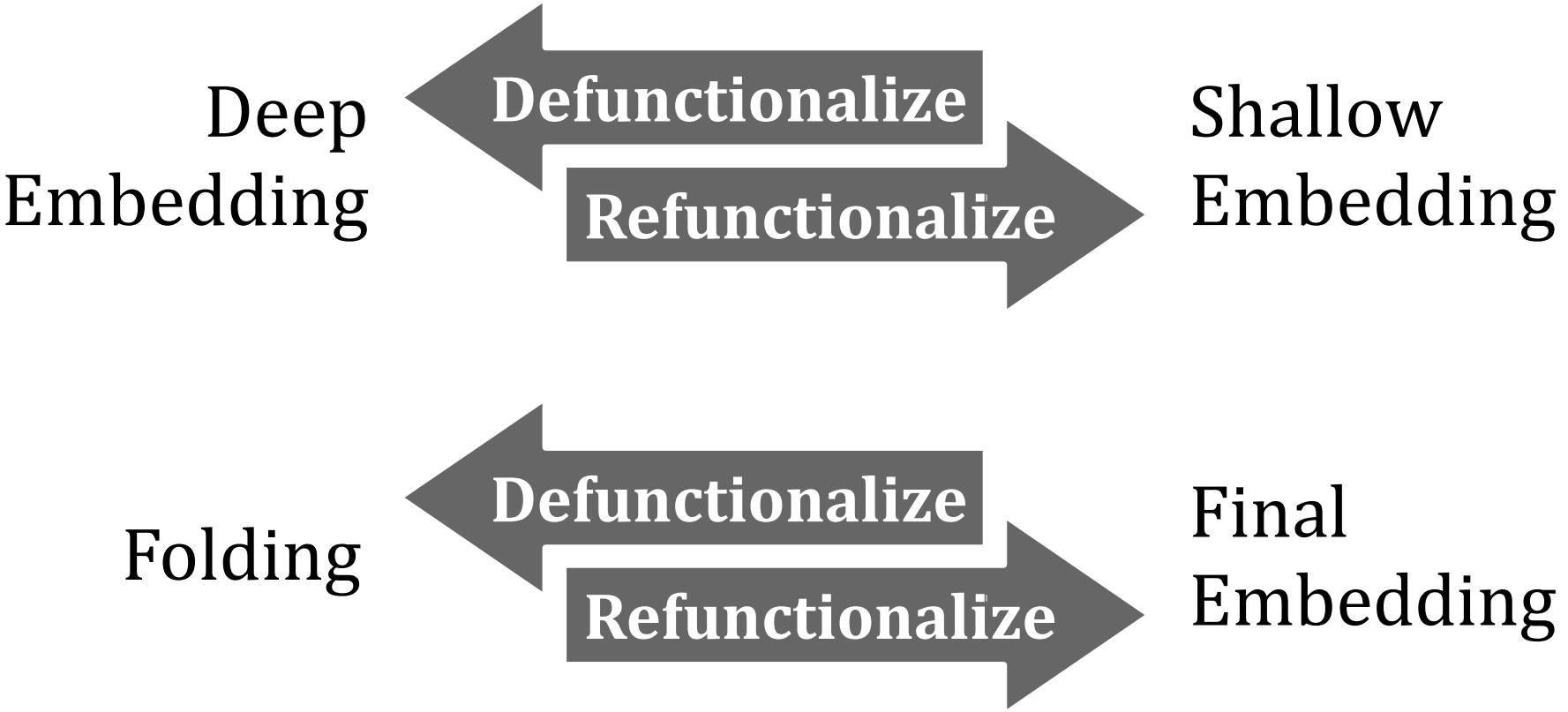
```
trait Exp { def eval: Int }

def Lit(n: Int) = new Exp { def eval = n }

def Add(l: Exp, r: Exp) =
  new Exp { def eval = l.eval + r.eval }
```

(Re|De)functionalization



Reynolds 1972 (Definitional Interpreters for Higher-Order Languages)
Danvy and Milliken 2007 (Refunctionalization at Work)

(Re|De)functionalization



Folding

```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def fold[T](e: Exp, alg: Alg[T]): T = e match {
  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
}
```

Folding

```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}
```

```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp
```

```
def fold[T](e: Exp, alg: Alg[T]): T = e match {
  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
}
```

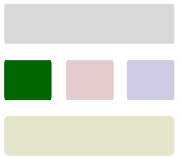
Folding

```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}
```

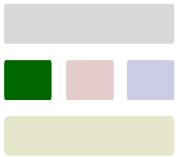
```
trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def fold[T](e: Exp, alg: Alg[T]): T = e match {
  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
}
```

Folding

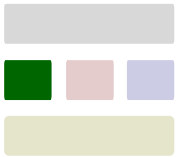
```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def fold[T](e: Exp, alg: Alg[T]): T = e match {
  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
}
```



```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp

def fold[T](e: Exp, alg: Alg[T]): T = e match {
  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
}
```

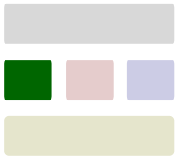


```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}
trait Exp { def fold[T](alg: Alg[T]): T }
case class Lit(n: Int) extends Exp
case class Add(l: Exp, r: Exp) extends Exp


  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
```



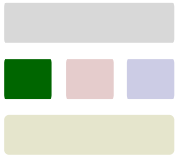
```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp { def fold[T](alg: Alg[T]): T }

case class Lit(n: Int) extends Exp

case class Add(l: Exp, r: Exp) extends Exp


  case Lit(n) => alg.Lit(n)
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
```

```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp { def fold[T](alg: Alg[T]): T }

def Lit(n: Int) = new Exp {
  def fold[T](alg: Alg[T]) = alg.lit(n)
}

case class Add(l: Exp, r: Exp) extends Exp

  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
```



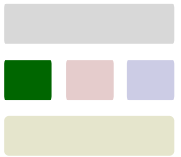
```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp { def fold[T](alg: Alg[T]): T }

def Lit(n: Int) = new Exp {
  def fold[T](alg: Alg[T]) = alg.lit(n)
}

case class Add(l: Exp, r: Exp) extends Exp
  case Add(l, r) => alg.Add(fold(l, alg), fold(r, alg))
```

```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp { def fold[T](alg: Alg[T]): T }

def Lit(n: Int) = new Exp {
  def fold[T](alg: Alg[T]) = alg.lit(n)
}

def Add(l: Exp, r: Exp) = new Exp {
  def fold[T](alg: Alg[T]) =
    alg.Add(l.fold(alg), r.fold(alg))
}
```

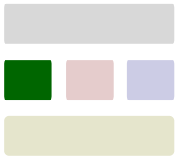


Final Embedding

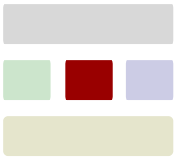
```
trait Alg[T] {
  def Lit(n: Int): T
  def Add(l: T, r: T): T
}

trait Exp { def fold[T](alg: Alg[T]): T }

def Lit(n: Int) = new Exp {
  def fold[T](alg: Alg[T]) = alg.lit(n)
}

def Add(l: Exp, r: Exp) = new Exp {
  def fold[T](alg: Alg[T]) =
    alg.Add(l.fold(alg), r.fold(alg))
}
```



To study the relationship of embedding approaches, we should study defunctionalization and refunctionalization



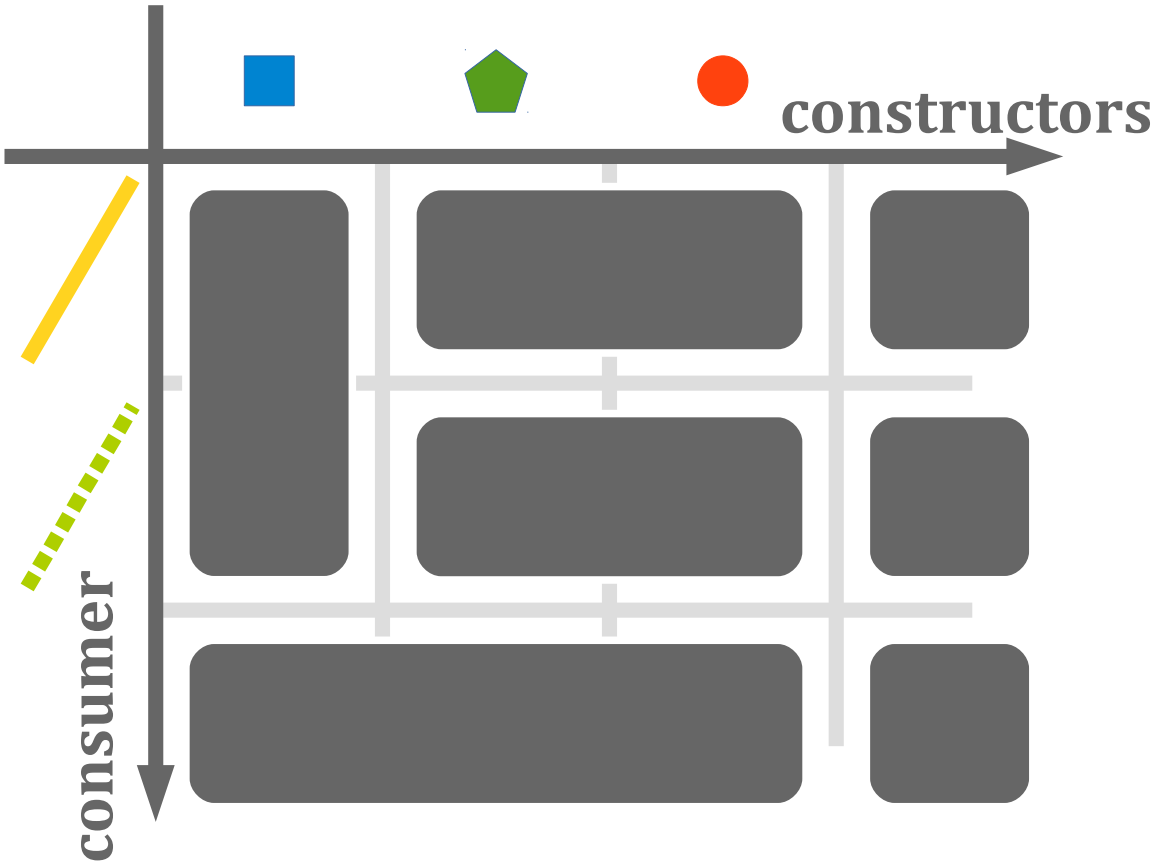
To study the relationship of embedding approaches, we should study defunctionalization and refunctionalization (for Scala-ish languages?!)

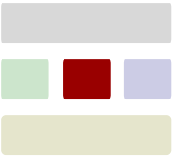


Modular, Scalable, and Compositional Encoding (of Attribute Grammars in Scala)

Rendel, Brachthäuser, Ostermann 2014 (From Object Algebras to ...)

Two Dimensions of Modularity



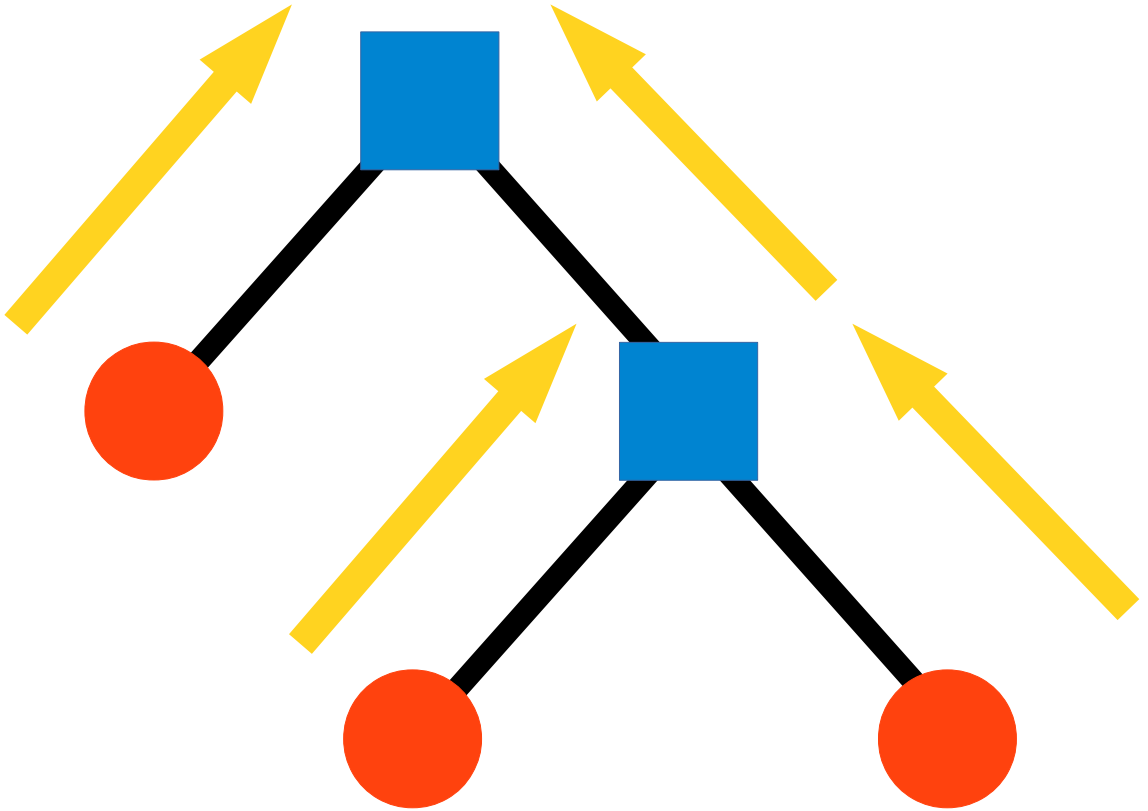


Third Dimension of Modularity

To scale to large programs, solutions to the expression problem need to support components with inner structure:

- Consumers that depend on other consumers.
- Consumers that depend on context information.
- Consumers that are fused together.

Bottom-Up Data Flow



Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1 .value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

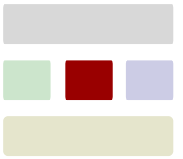
e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```



Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n \quad \{ \text{Lit} \}$

$e_1 \rightarrow e_2 \text{ "+" } e_3 \quad \{ \text{Add} \}$

Equations

$e_0.\text{value} = n$

$e_1.\text{value} = e_2.\text{value} + e_3.\text{value}$

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

Signature

```
trait Sig[E] {
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int] {
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Synthesized Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_0.value = n

e_1.value = e_2.value + e_3.value

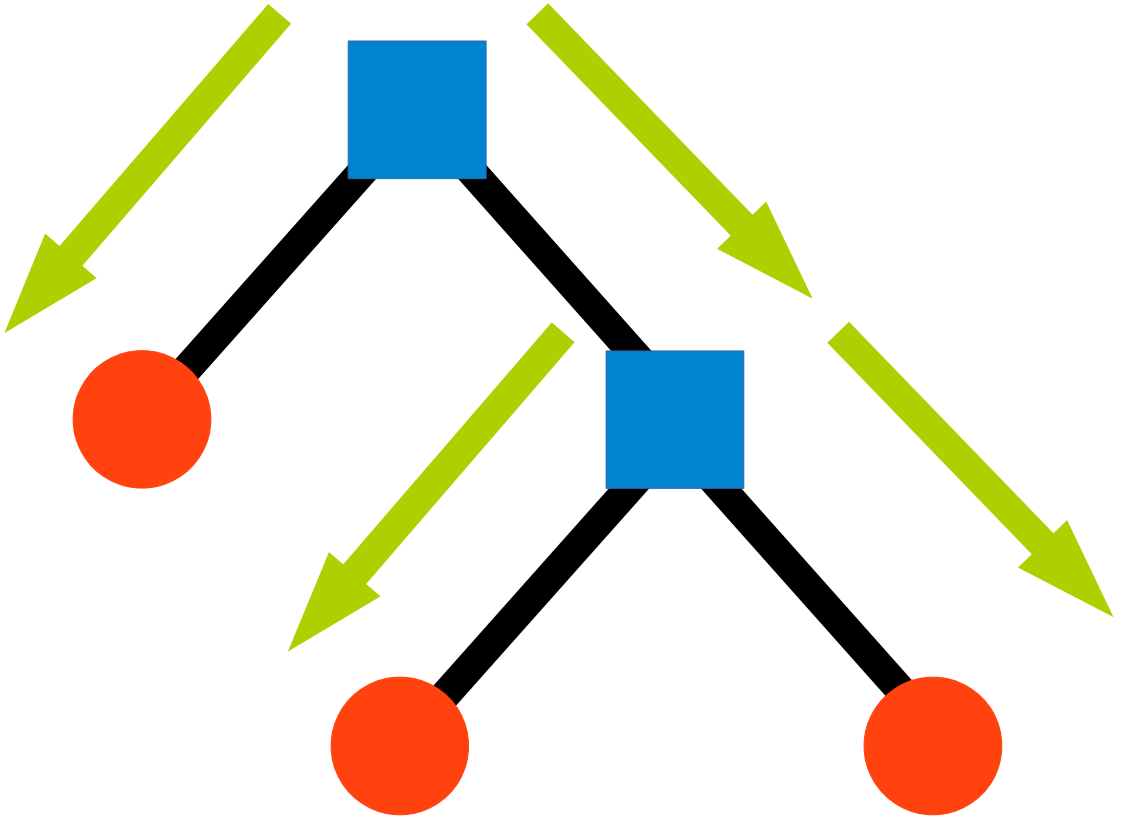
Signature

```
trait Sig[E]{
  def Lit: Int ⇒E
  def Add: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Int]{
  def Lit = n ⇒n
  def Add = (e2, e3) ⇒e2 + e3
}
```

Top-Down Data Flow



Inherited Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_2.left = **true**

e_3.left = **false**

Signature

```
trait Sig[E] {
  def Add1: E ⇒E
  def Add2: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Bool] {
  def Add1 = e ⇒true
  def Add2 = (e1, e2) ⇒false
}
```

Inherited Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_2.left = **true**

e_3.left = **false**

Signature

```
trait Sig[E] {
  def Add1: E ⇒E
  def Add2: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Bool] {
  def Add1 = e ⇒true
  def Add2 = (e1, e2) ⇒false
}
```

Inherited Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ "+" e_3 { Add }

Equations

e_2 .left = **true**

e_3 .left = **false**

Signature

```
trait Sig[E] {
  def Add1: E ⇒E
  def Add2: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Bool] {
  def Add1 = e ⇒true
  def Add2 = (e1, e2) ⇒false
}
```

Inherited Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ '+' e_3 { Add }

Equations

e_2.left = **true**

e_3.left = **false**

Signature

```
trait Sig[E] {
  def Add1: E ⇒E
  def Add2: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Bool] {
  def Add1 = e ⇒true
  def Add2 = (e1, e2) ⇒false
}
```

Inherited Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ '+' e_3 { Add }

Equations

e_2.left = **true**

e_3.left = **false**

Signature

```
trait Sig[E] {
  def Add1: E ⇒E
  def Add2: (E, E) ⇒E
}
```

Algebra

```
val Alg = new Sig[Bool] {
  def Add1 = e ⇒true
  def Add2 = (e1, e2) ⇒false
}
```

Inherited Attributes

Grammar

$e_0 \rightarrow n$ { Lit }

$e_1 \rightarrow e_2$ '+' e_3 { Add }

Signature

```
trait Sig[E] {
  def Add1: E ⇒E
  def Add2: (E, E) ⇒E
}
```

Equations

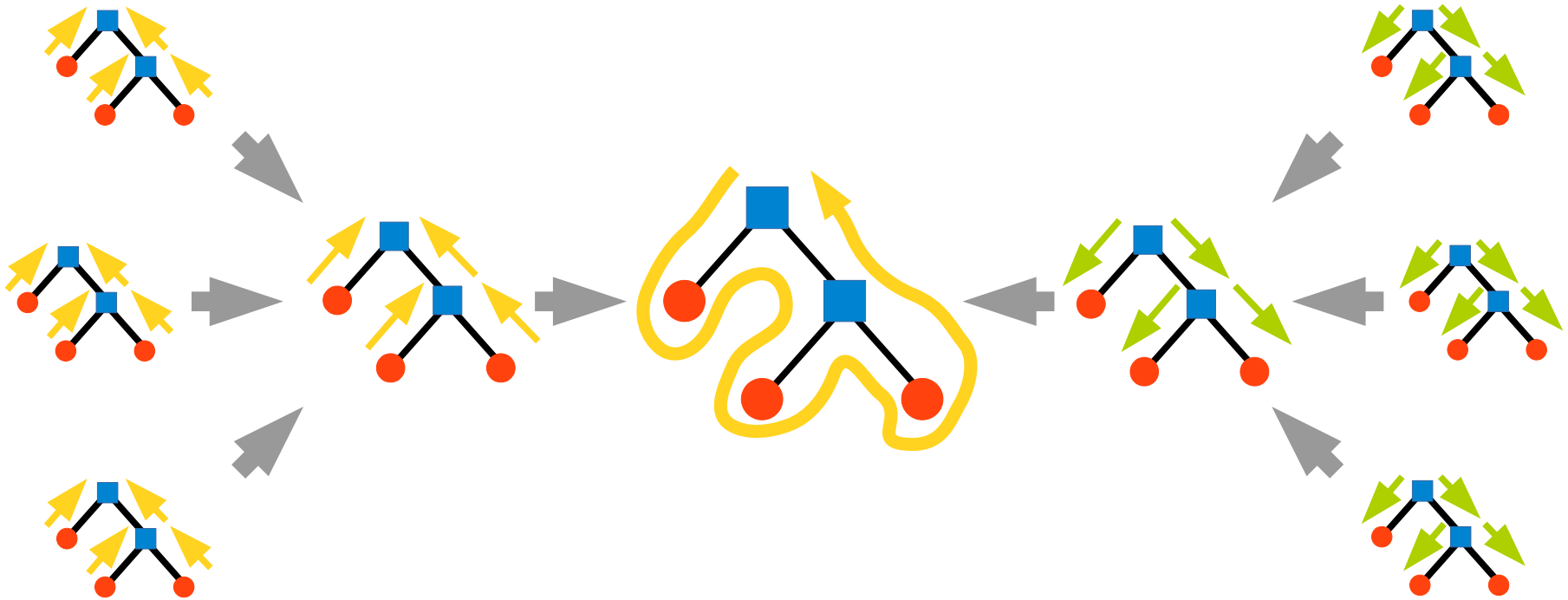
e_2.left = **true**

e_3.left = **false**

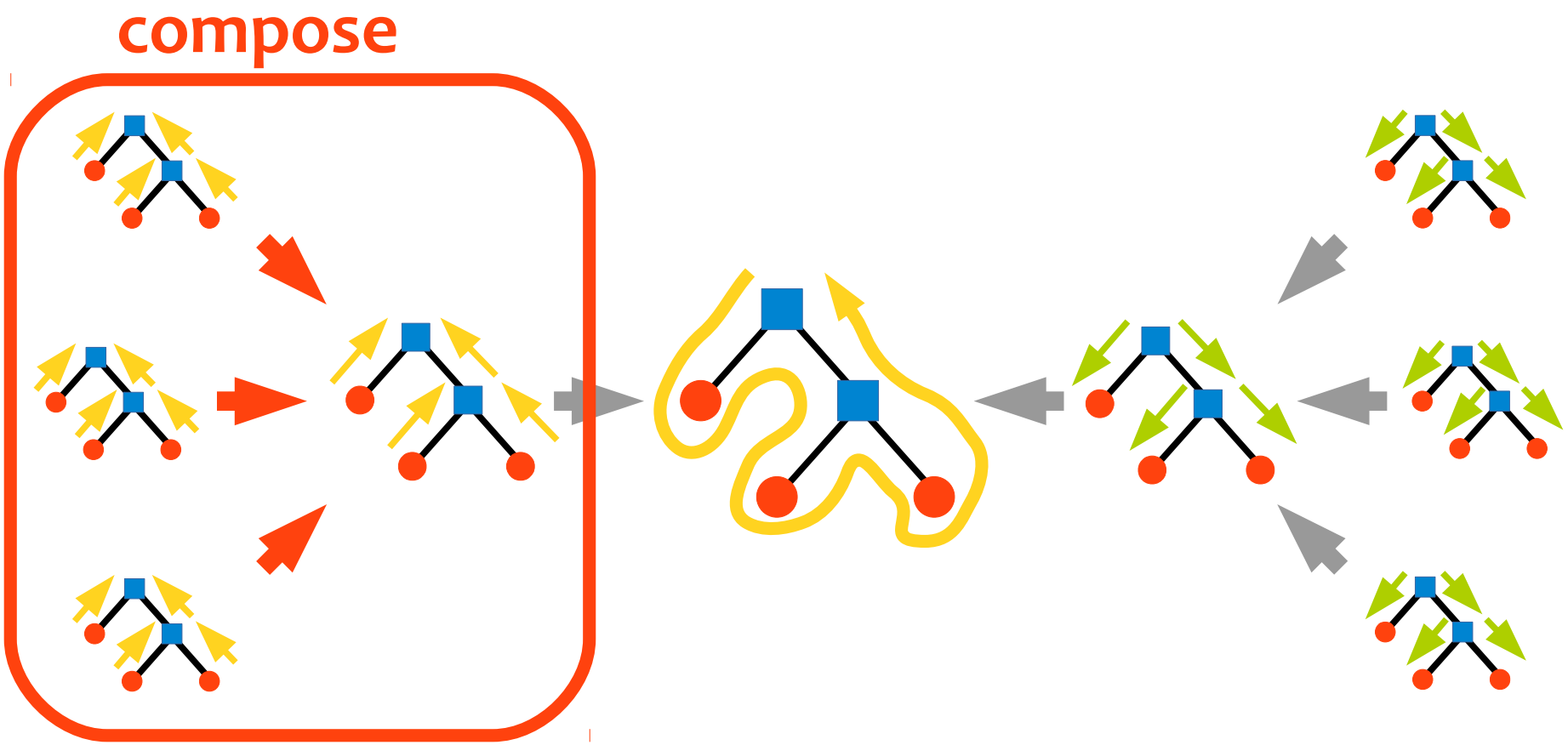
Algebra

```
val Alg = new Sig[Bool] {
  def Add1 = e ⇒true
  def Add2 = (e1, e2) ⇒false
}
```

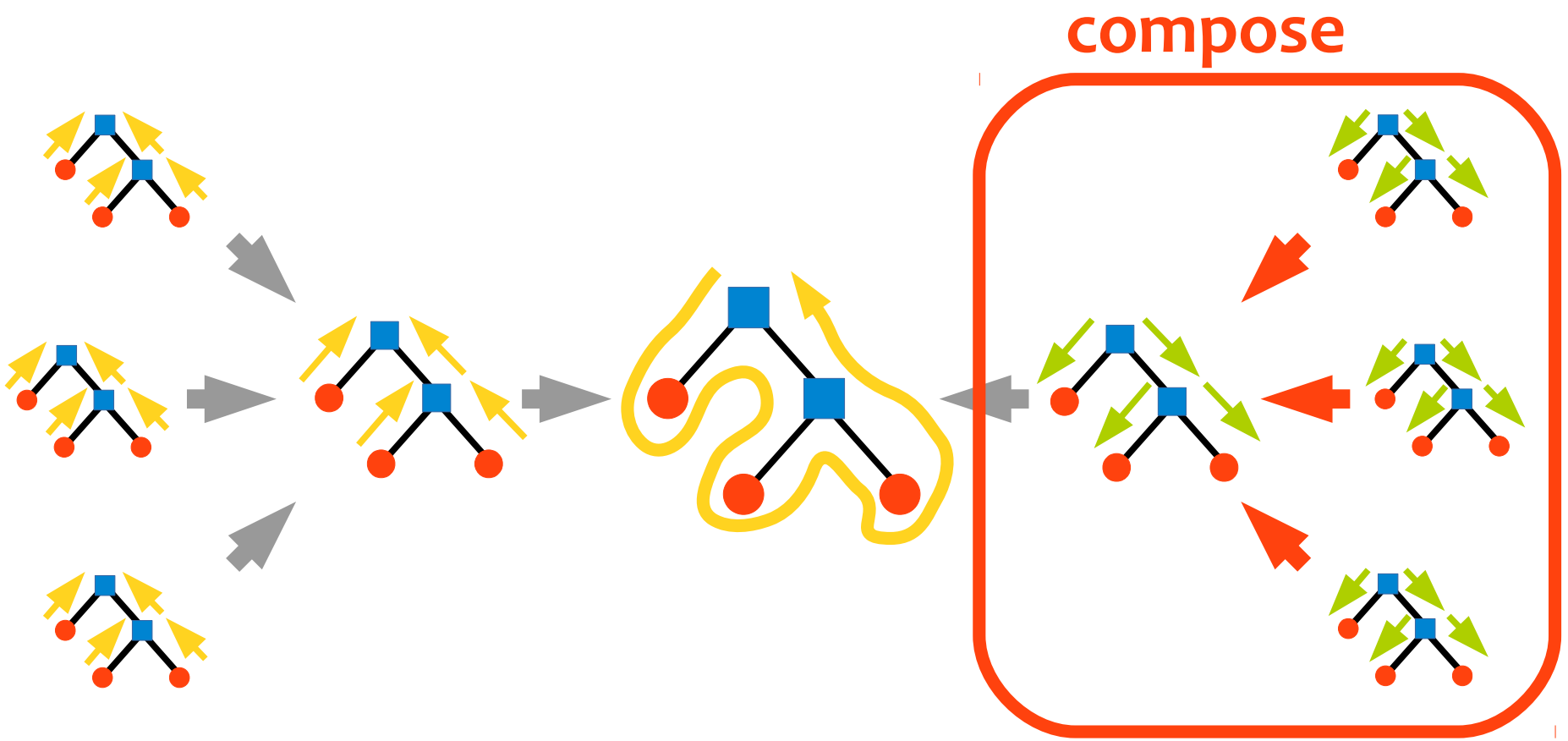
Composition



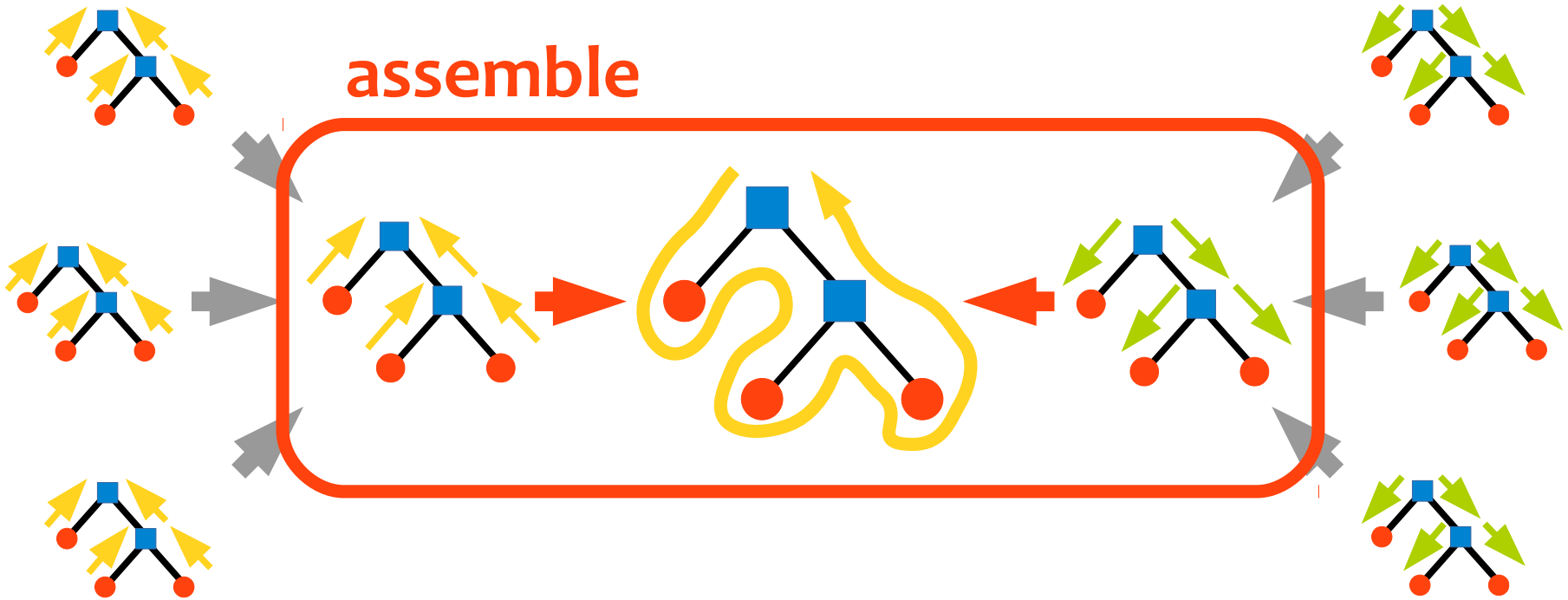
Composition

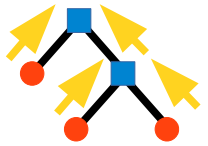
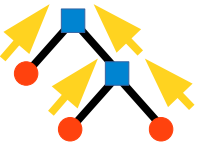
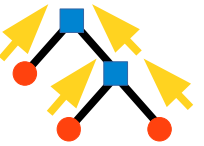


Composition



Composition



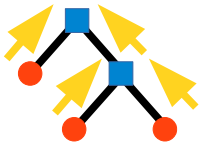
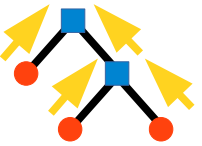
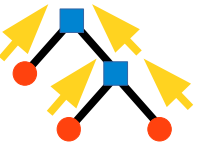
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

compose( , ) = 

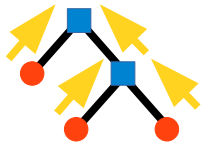
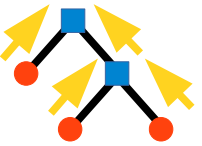
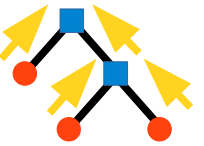
Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
```

```
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

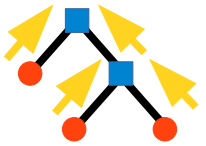
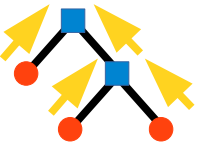
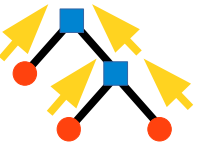
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

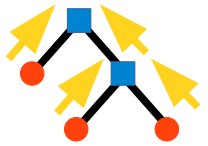
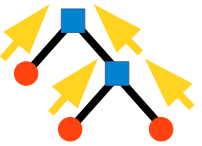
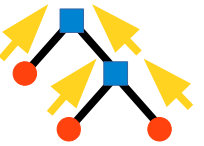
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

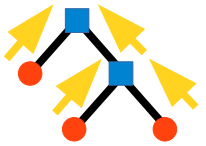
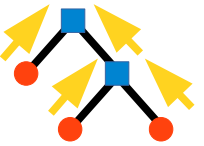
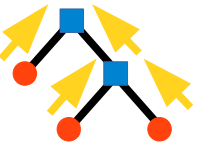
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

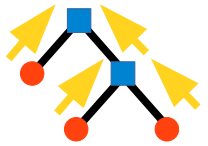
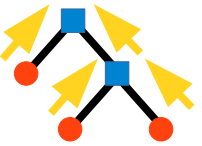
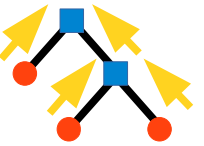
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

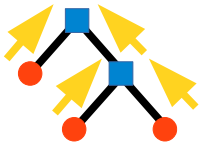
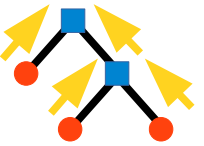
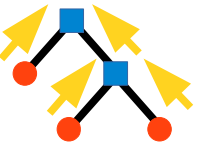
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) => A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

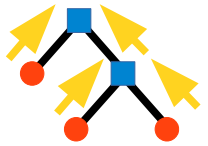
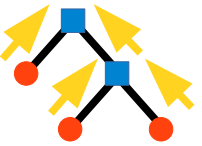
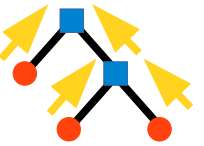
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

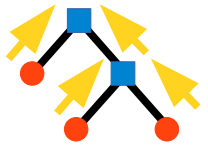
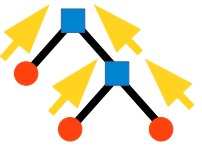
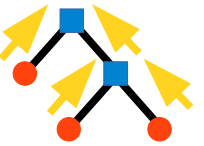
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

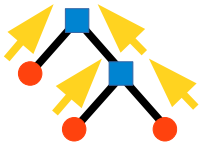
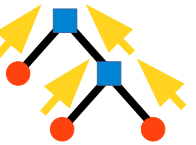
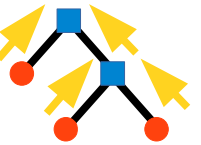
compose( , ) = 

Extensible Records

```
trait HasValue {def value: Int}
trait HasLeft {def left: Bool}
def mix[A, B](implicit m: Mix[A, B]): (A, B) ⇒A with B
```

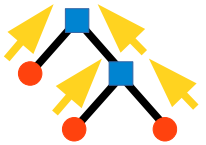
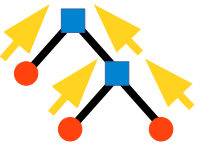
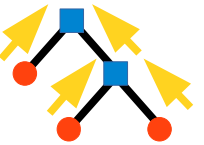
Implicit Macros

```
implicit def mixAll[A, B]: Mix[A, B] = macro impl[A, B]
def impl[A, B](c: Context)(implicit aT: c.WeakTypeTag[A],
bT: c.WeakTypeTag[B]): c.Expr[Mix[A, B]] = { ...
```

compose( , ) = 

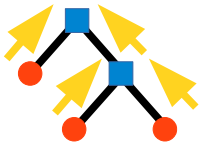
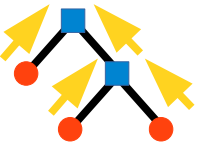
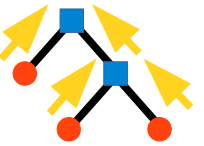
Dependency Tracking

```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

compose( , ) = 

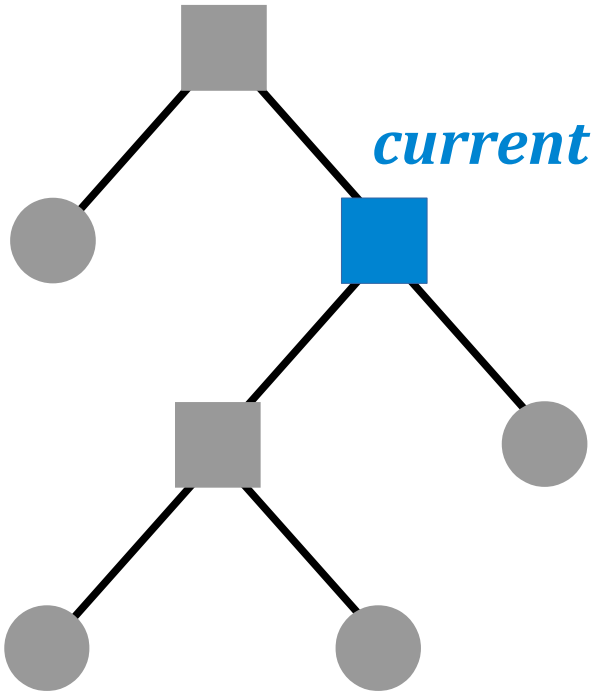
Dependency Tracking

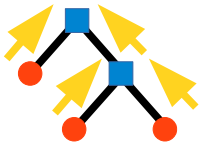
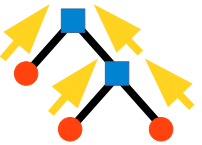
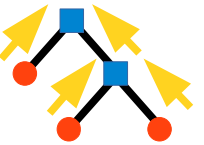
```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

compose( , ) = 

Dependency Tracking

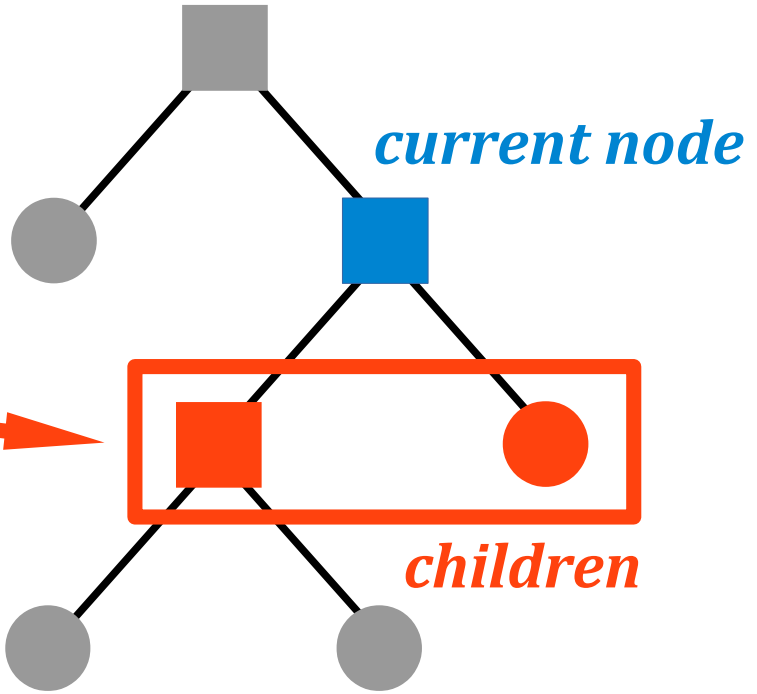
```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

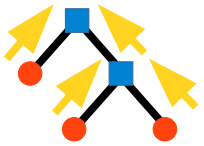
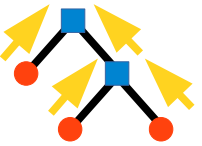
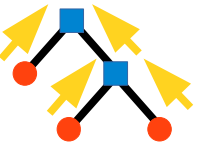


compose( , ) = 

Dependency Tracking

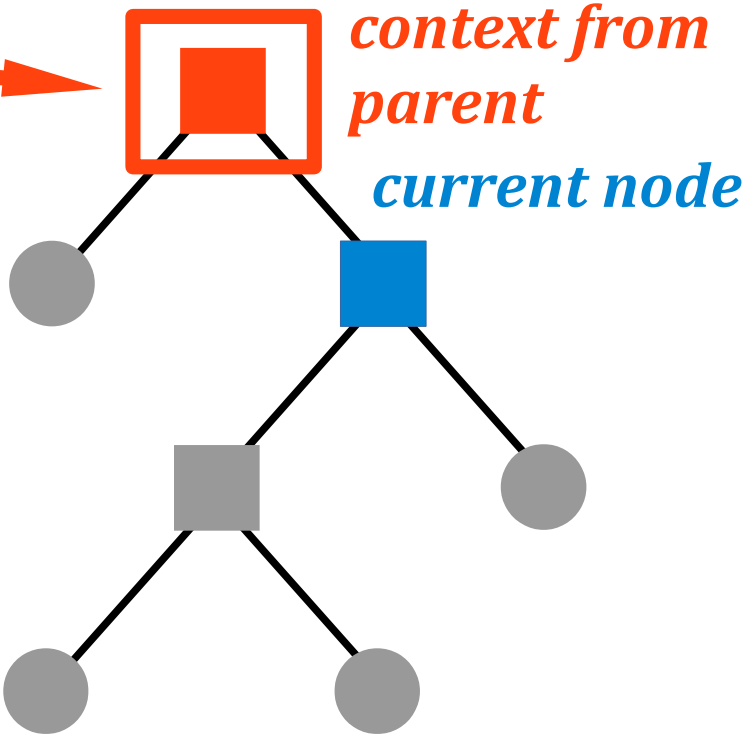
```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

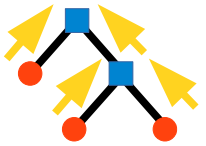
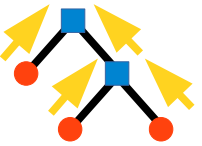
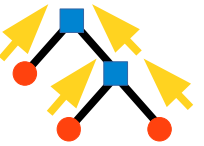


compose( , ) = 

Dependency Tracking

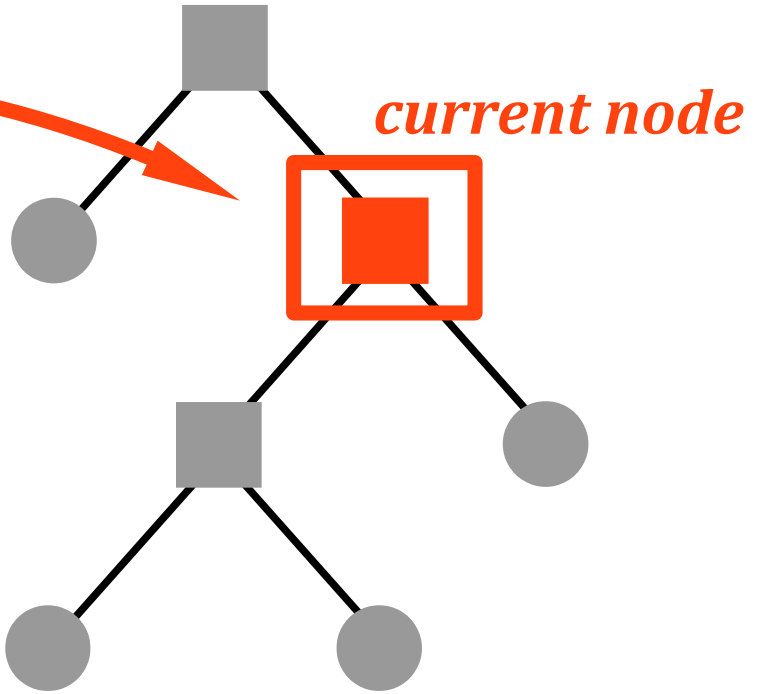
```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

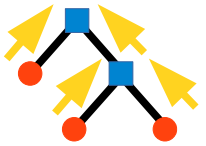
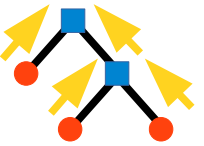
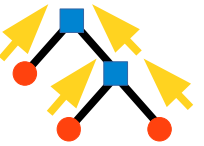


compose( , ) = 

Dependency Tracking

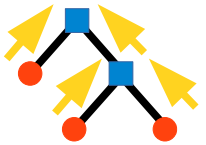
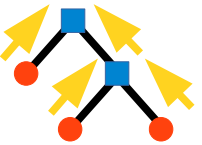
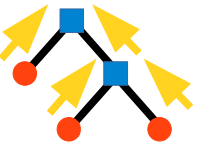
```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```



compose( , ) = 

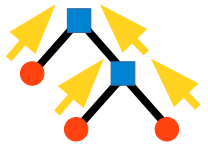
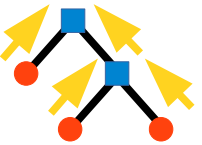
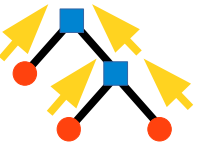
Dependency Tracking

```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

compose( , ) = 

Dependency Tracking

```
trait Sig[-E, -C, +O] {
  def Lit: Int ⇒C ⇒O
  def Add: (E, E) ⇒C ⇒O
}
```

compose( , ) = 

Composing two algebras

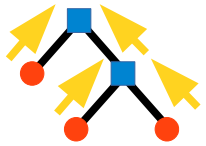
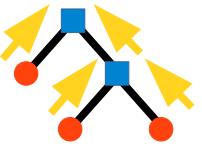
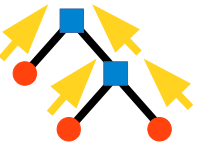
def compose

[E_1, C_1, O_1, E_2, C_2 >: C_1 **with** O_1, O_2]

(alg_1: Sig[E_1, C_1, O_1],

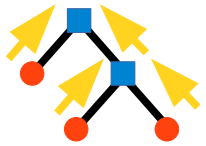
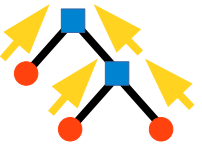
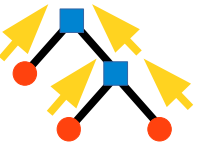
alg_2: Sig[E_2, C_2, O_2]):

Sig[E_1 **with** E_2, C_1, O_1 **with** O_2]

compose( , ) = 

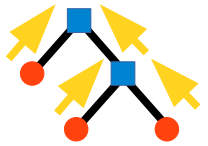
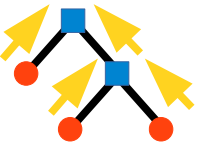
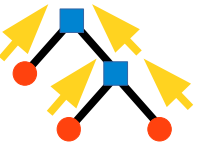
Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
              Sig[E1 with E2, C1, O1 with O2]
```

compose( , ) = 

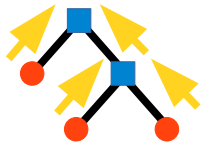
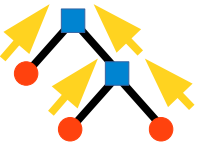
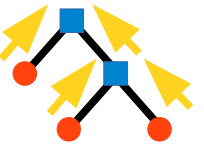
Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
              Sig[E1 with E2, C1, O1 with O2]
```

compose( , ) = 

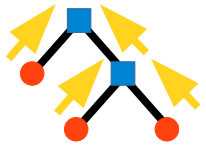
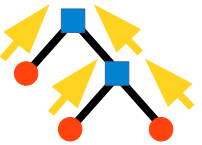
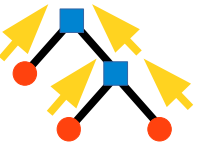
Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
             Sig[E1 with E2, C1, O1 with O2]
```

compose( , ) = 

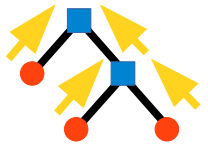
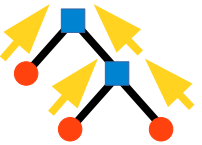
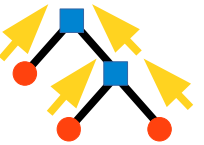
Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
            Sig[E1 with E2, C1, O1 with O2]
```

compose( , ) = 

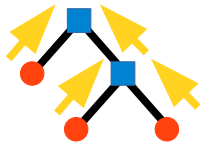
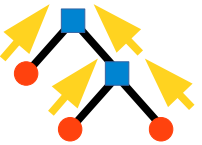
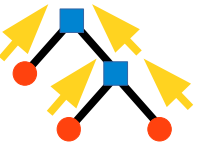
Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
             Sig[E1 with E2, C1, O1 with O2]
```

compose( , ) = 

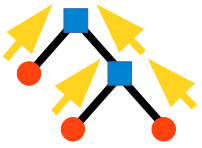
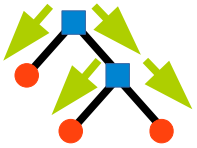
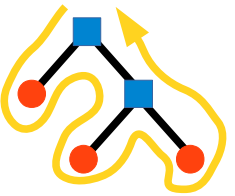
Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
              Sig[E1 with E2, C1, O1 with O2]
```

compose( , ) = 

Composing two algebras

```
def compose
        [E1, C1, O1, E2, C2 >: C1 with O1, O2]
        (alg1: Sig[E1, C1, O1],
         alg2: Sig[E2, C2, O2]):
             Sig[E1 with E2, C1 O1 with O2]
```

assemble( , ) = 

Assembling a one-pass traversal

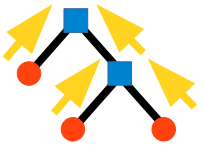
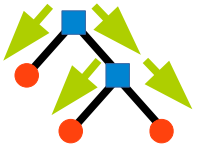
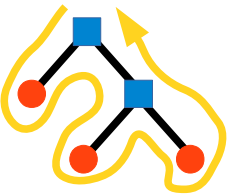
def assemble

[C, O]

(alg_1: Sig_1[C with O, C, O],

alg_2: Sig_2[C with O, C, C]):

Sig[C ⇒C with O]

assemble( , ) = 

Assembling a one-pass traversal

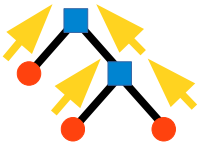
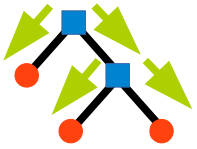
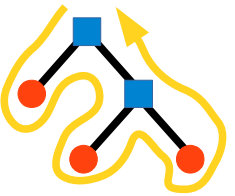
def assemble

[C, O]

(alg$_1$: Sig$_1$[C with O, C, O],

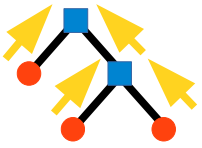
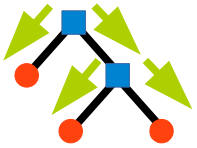
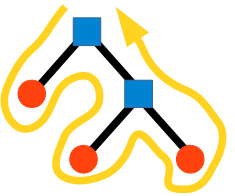
alg$_2$: Sig$_2$[C with O, C, C])

Sig[C ⇒C with O]

assemble( , ) = 

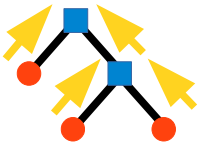
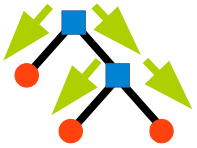
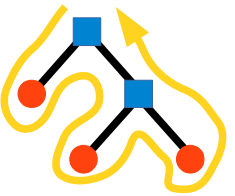
Assembling a one-pass traversal

```
def assemble
        [C, O]
        (alg1: Sig1[C with O, C, O],
         alg2: Sig2[C with O, C, C]):
            Sig[C ⇒C with O]
```

assemble( , ) = 

Assembling a one-pass traversal

```
def assemble
        [C, O]
        (alg1: Sig1[C with O, C, O],
         alg2: Sig2[C with O, C, C]):
            Sig[C ⇒C with O]
```

assemble( , ) = 

Assembling a one-pass traversal

def assemble

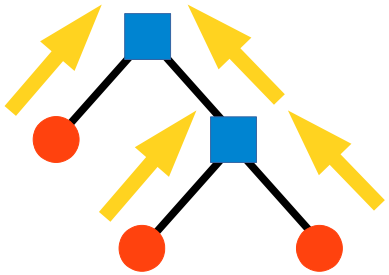
[C, O]

(alg_1: Sig_1[C with O, C, O],

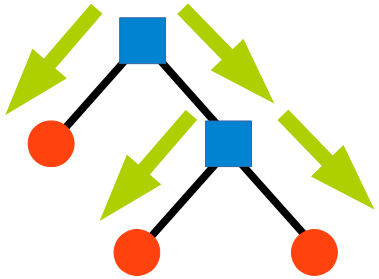
alg_2: Sig_2[C with O, C, C]):

Sig[C ⇒C with O]

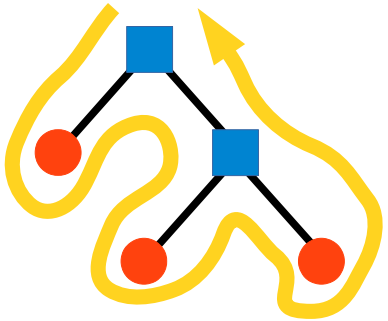
Results



Object algebras correspond
to **synthesized attributes**
(bottom-up data-flow)

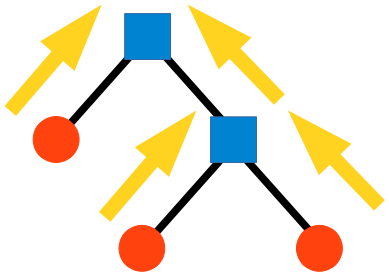


We **extend** object algebras
to support **inherited attributes**
(top-down data flow)

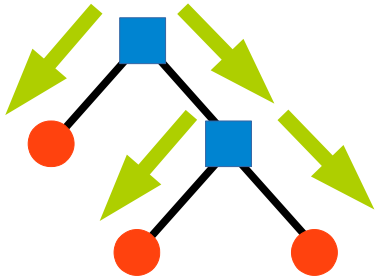


We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

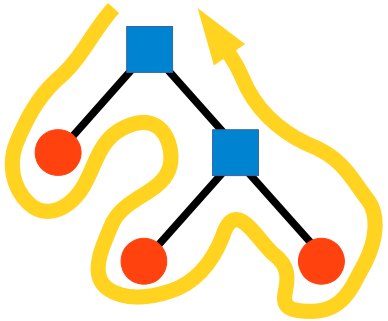
Results



Object algebras correspond
to **synthesized attributes**
(bottom-up data-flow)

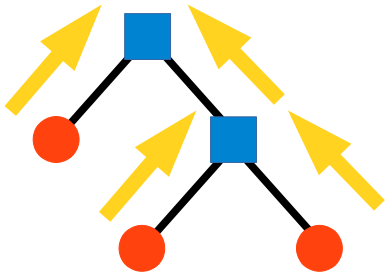


We **extend** object algebras
to support **inherited attributes**
(top-down data flow)

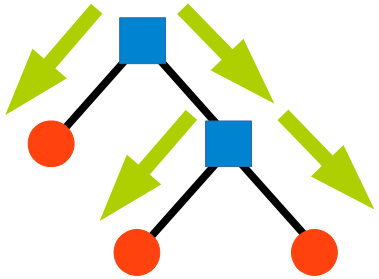


We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

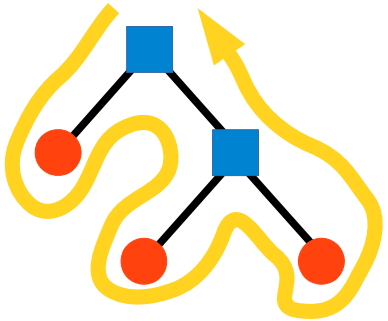
Results



Object algebras correspond
to **synthesized attributes**
(bottom-up data-flow)

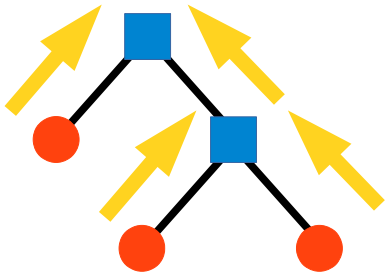


We **extend** object algebras
to support **inherited attributes**
(top-down data flow)

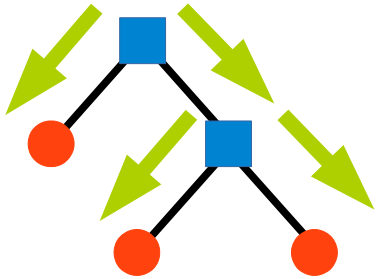


We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

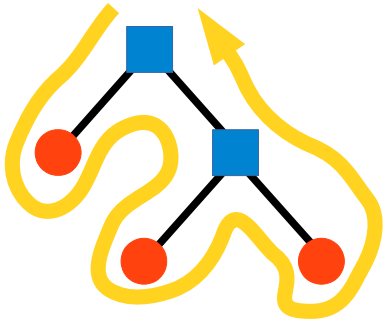
Results



Object algebras correspond
to **synthesized attributes**
(bottom-up data-flow)



We **extend** object algebras
to support **inherited attributes**
(top-down data flow)



We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

Modularizing a One-Pass Compiler

- existing one-pass compiler for a subset of C
- 9 nonterminals
- written for teaching at Aarhus university (not by the authors)

Monolithic compiler

1 file
807 lines of Java code
entangled

```
static void parseFunction(Map funcs, Map prototypes) {
  Map vars = new HashMap();
  String name;
  checkToken(tINT);
  skipToken(tID);
  name = tldValue;
  skipToken(tLPAR);
  int args = parseFormals(vars);
  skipToken(tRPAR);
  skip();
  if (tKind==tSEMI) {
    nextToken();
    if (prototypes.containsKey(name))
      compileError("duplicate declaration of "+name);
    if (funcs.containsKey(name) && args!=funcs.get(name))
      compileError("conflicting declaration of "+name);
    prototypes.put(name,args);
  } else {
    if (funcs.containsKey(name))
      compileError("duplicate implementation of "+name);
    if (prototypes.containsKey(name) && args!=prototypes.get(name))
      compileError("conflicting implementation of "+name);
    funcs.put(name,args);
    code(".method "+name);
    code(".args "+(args+1));
    parseBody(args,vars,funcs,prototypes);
    code("bipush 0");
    code("ireturn");
  }
}
```

Modularized compiler

ca. 25 files
1620 lines of Scala code
modular

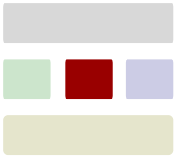
```
def parse(attr: IA)(implicit s: TokenStream): IA with SA = {
  if (s is tInt) s eat tInt;
  s skip()
  val id = s.eatId().value
  s skip tLPar
  val formals = parseFormals
  s skip tRPar
  s skip()
  if (s is tSemi) {
    s eat tSemi;
    compose(attr, sys.funDecl(id, formals, attr))
  } else {
    s skip tLBrace
    val funDefStmIA = ida.funDefStmIA(attr, id, formals)
    val funDefInIA2 = decls.parse(funDefInIA)
    val funDefInIA = ida.funDefStateCattr, id, formals, funDefInIA2)
    val funDefAllA = stmts.parse(funDefInIA)
    s skip tRBrace
    compose(attr, sys.funDef(id, formals, funDefInAll, funDefStmAll, attr))
  }
}
```

```
object FunctionVariables extends Function.Syn[Any, HasVariables, HasVariables, Any, HasVariables with HasOffset, HasVariables] {
  def funDefDecls(parent: IA, id: String, formals: List[String]): DeclsIA =
    parent.variables := formals.zipWithIndex.map {
      case (a, ida) => (a, ida + param1.offset)
    }.toMap
  def funVariables(parent: IA, id: String, formals: List[String], decls: Decls): StmtsIA =
    decls
}
```

```
object FunctionPrototypes extends Function.Syn.Complete[Any, Any, HasPrototypes, HasPrototypes] {
  def funDecl(id: String, formals: List[String], self: Self): Out =
    if (self.prototypes isDefinedAt id) {
      sys error (s"duplicate declaration of $id")
    } else {
      HasPrototypes(self.prototypes + (id -> formals))
    }
  def funDef(id: String, formals: List[String], decls: Decls, stmts: Stmts, self: Self): Out =
    if (self.prototypes isDefinedAt(id) && self.prototypes(id) != formals) {
      sys error (s"conflicting implementation of $id")
    } else {
      HasPrototypes(self.prototypes)
    }
}
```

```
object FunctionFuncs extends Function.Syn.Complete[Any, Any, HasFuncs, HasFuncs] {
  def funDecl(id: String, formals: List[String], self: Self): Out =
    if (self.funcs isDefinedAt(id) && self.funcs(id) != formals) {
      sys error (s"conflicting declaration of $id")
    } else {
      HasFuncs(self.funcs)
    }
  def funDef(id: String, formals: List[String], decls: Decls, stmts: Stmts, self: Self): Out =
    if (self.funcs isDefinedAt id) {
      sys error (s"duplicate implementation of $id")
    } else {
      HasFuncs(self.funcs + (id -> formals))
    }
}
```

```
object FunctionCode extends Function.Syn.CompleteUseCode with HasCount, HasCode, HasCode, Any] {
  def funDecl(id: String, formals: List[String], self: Self): Out = Seq.empty
  def funDef(id: String, formals: List[String], decls: Decls, stmts: Stmts, self: Self): Out = {
    val localsCode = if (decls.count > 0) Seq(s".locals ${decls.count}") else Seq.empty
    Seq(
      s".method $id",
      s".args ${formals.size + 1}"
    ) ++ localsCode ++ decls.code ++ stmts.code ++ Seq(
      "bipush 0",
      "ireturn"
    )
  }
}
```



Properties of the Encoding

Modular
Attributes are defined and type-checked separately

Scalable
Scala code size is linear in AG specification size.

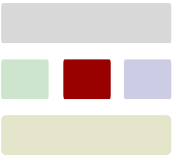
Compositional
Each AG artifact is represented as a Scala value.

Properties of the Encoding

Modular
Attributes are defined and type-checked separately

Scalable
Scala code size is linear in AG specification size.

Compositional
Each AG artifact is represented as a Scala value.



Properties of the Encoding

Modular

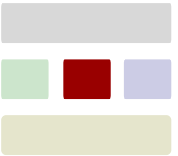
Attributes are defined and type-checked separately

Scalable

Scala code size is linear in AG specification size.

Compositional

Each AG artifact is represented as a Scala value.

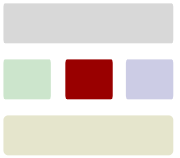


Properties of the Encoding

Modular
Attributes are defined and type-checked separately

Scalable
Scala code size is linear in AG specification size

Compositional
Each AG artifact is represented as a Scala value.



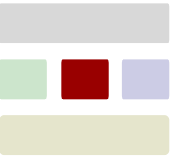
Scala Experience

Good

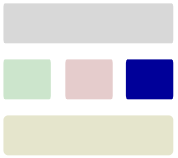
- Traits & static mixin composition
- Type Inference (when it works)
- Implicits
- Existence of Macros
- Implicit Macros

Bad

- Missing introduction for intersection types
- Type Inference (when it fails)
- Details of macro programming

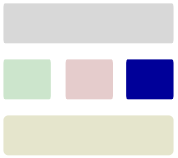


Some OO-inspired features of Scala are very good for modular, scalable and compositional encoding, but we don't need full OO.



The Co-Expression Problem

based on work in progress with
Brachthäuser, Giarrusso and Ostermann

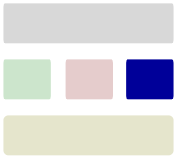


Duality

A **data type** is defined
by **constructors**.

Functions **consume**
data values by
covering all **constructors**.

How to extend
constructors and
consumers?



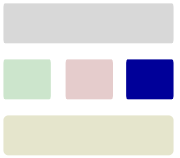
Duality

A **data type** is defined by **constructors**.

Functions **consume** **data** values by covering all **constructors**.

How to extend **constructors** and **consumers**?

A **codata type** is defined by **destructors**.



Duality

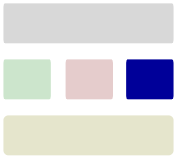
A **data type** is defined by **constructors**.

Functions **consume** **data** values by covering all **constructors**.

How to extend **constructors** and **consumers**?

A **codata type** is defined by **destructors**.

Functions **generate** **codata** values by covering all **destructors**.



Duality

A **data type** is defined by **constructors**.

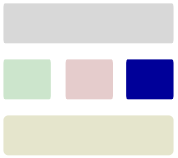
Functions **consume data** values by covering all **constructors**.

How to extend **constructors** and **consumers**?

A **codata type** is defined by **destructors**.

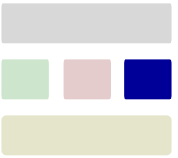
Functions **generate codata** values by covering all **destructors**.

How to extend **destructors** and **generators**?



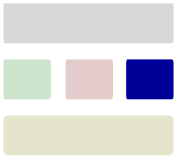
Impact

- Where are real instances of the co-expression problem?
- Can (should?) the co-expression problem guide research on codata representation like the expression problem guided research on data representation?



Solutions

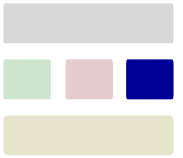
- Which solutions of the expression problem also solve the co-expression problem?
- Which solutions of the expression problem are dualizable to solutions of the co-expression problem?
- Are there solutions that are unique to one of the problems, maybe pointing to asymetric support for data and codata in programming languages?



Expression Lemma

- Seems to formalize the semantic aspects of a combination of the usual expression problem and the co-expression problem.
- In what setting should we study the syntactic and engineering aspects?

Lämmel, Rypacek 2008 (The Expression Lemma)



To investigate the relationship of expression problem and coexpression problem we should use a language with symmetric support for data and codata types.

A Very Simple Language with Symmetric Support for Data and Codata

Rendel, Trieflinger, Ostermann 2015 (Automatic Refunctionalization ...)

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), n) = n
  add(succ(m), n) =
    succ(add(m, n))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(n).head() = n
  count(n).tail() =
    count(succ(n))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Symmetric Support

data Nat **where**
  zero() : Nat
  succ(Nat) : Nat

fun add(Nat, Nat) **were**
  add(zero(), *n*) = *n*
  add(succ(*m*), *n*) =
    succ(add(*m*, *n*))

codata Stream **where**
  Stream.head() : Nat
  Stream.tail() : Stream

fun count(Nat) **where**
  count(*n*).head() = *n*
  count(*n*).tail() =
    count(succ(*n*))

Abel, Pientka, Thibodeau, Setzer 2013 (Copatterns: Programming …)

Very Simple

- Purely functional.
- Simple types.
- No first-class functions.
- No case expressions.
- No cocase expressions.
- No local variables.

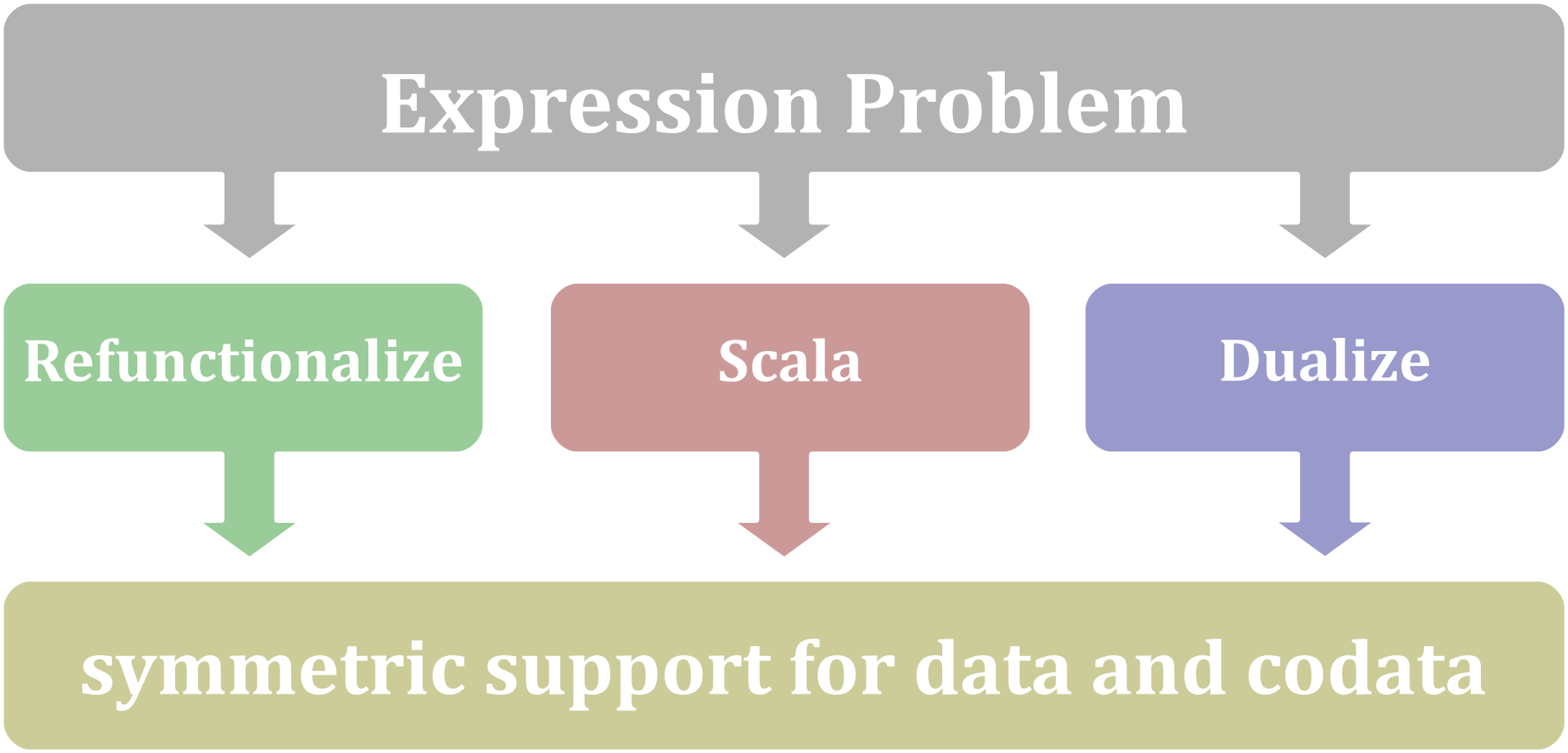
First Results

- Support for full defunctionalization and refunctionalization.
- Programs can be written as matrices, so that (de|re)functionalization both correspond to matrix transposition.
- These matrices look like the matrices from the expression and coexpression problems.

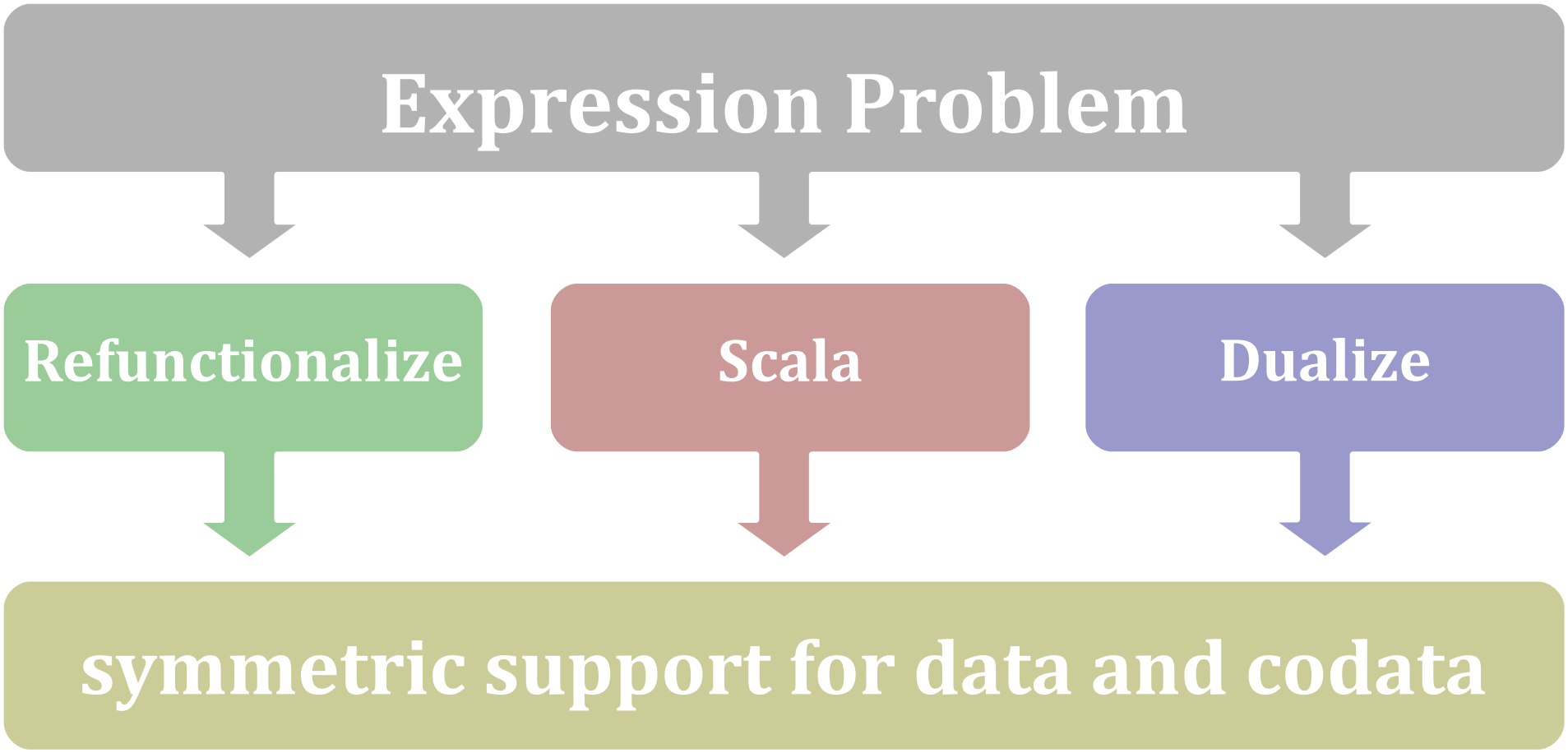
Automatic Refunctionalization To a Language with Copattern Matching With Applications to the Expression Problem

Presentation at ICFP
Tuesday, 4pm

Summary


Expression Problem
Refunctionalize
Scala
Dualize
symmetric support for data and codata

Summary



Thanks