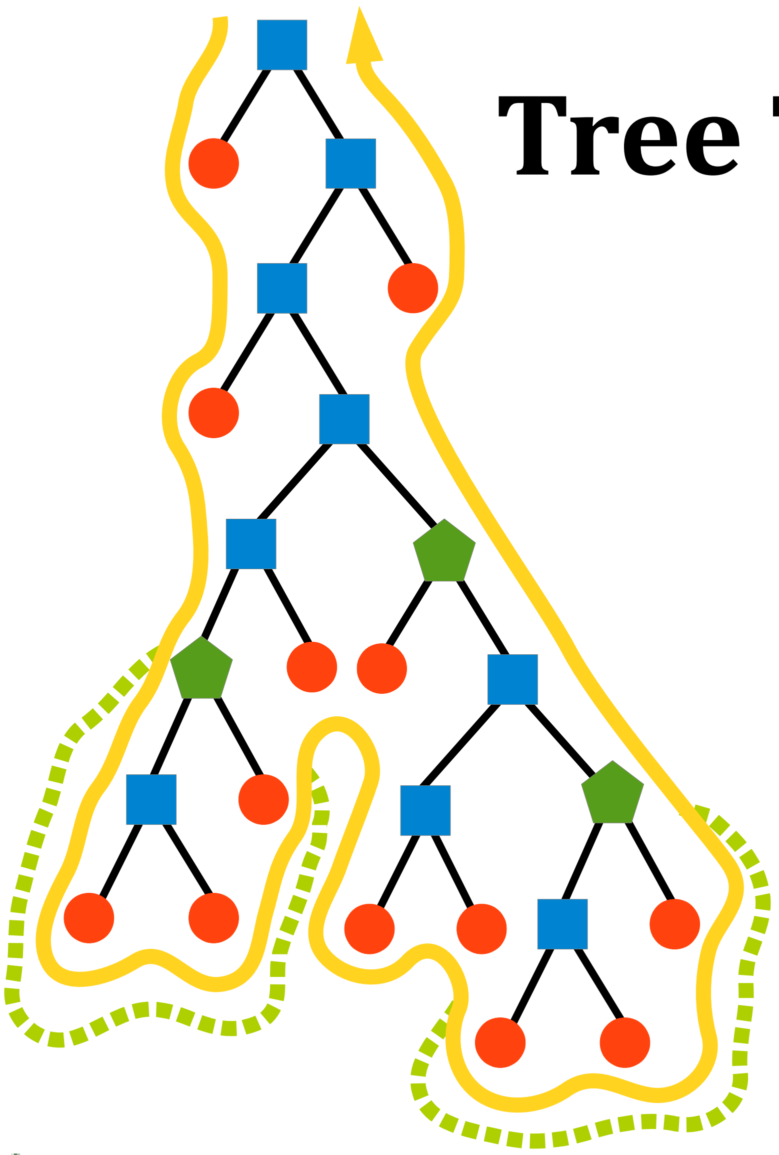
From Object Algebras to Attribute Grammars

Tillmann Rendel · Jonathan Brachthäuser · Klaus Ostermann
University of Marburg · University of Tübingen

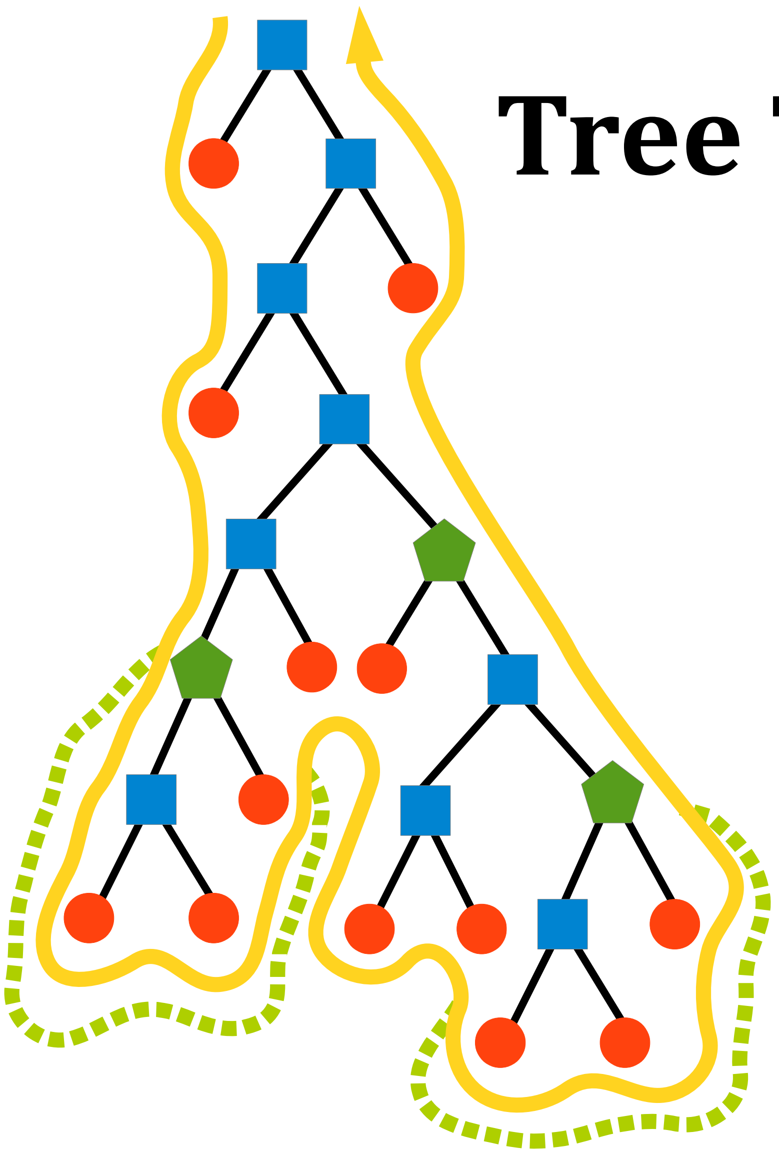
http://www.informatik.uni-marburg.de/~rendel/oa2ag

Presentation by Tillmann Rendel at the International Conference
on Object-Oriented Programming, Systems, Languages, and Applications
Portland, Oregon, October 23, 2014

Tree Traversals

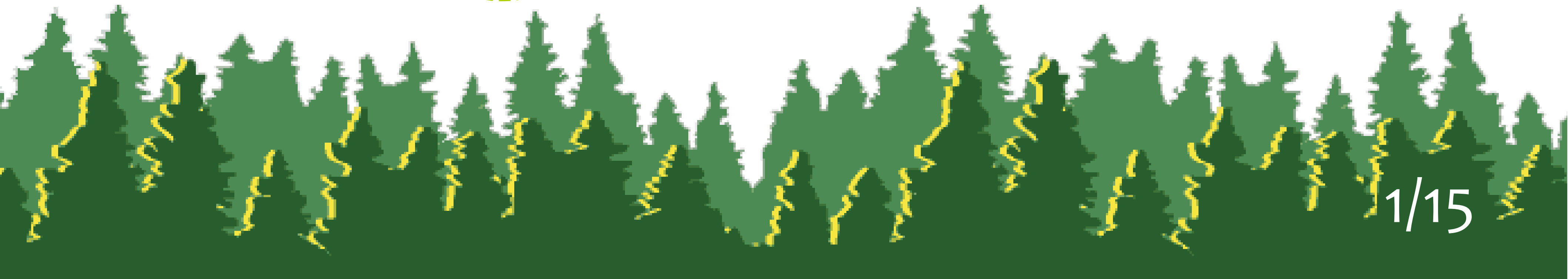


Tree Traversals



Tree Traversals

How to structure
a program
that contains
multiple traversals
of complex trees?



Folds & Traversal Schemes

in functional programming

Visitor Pattern

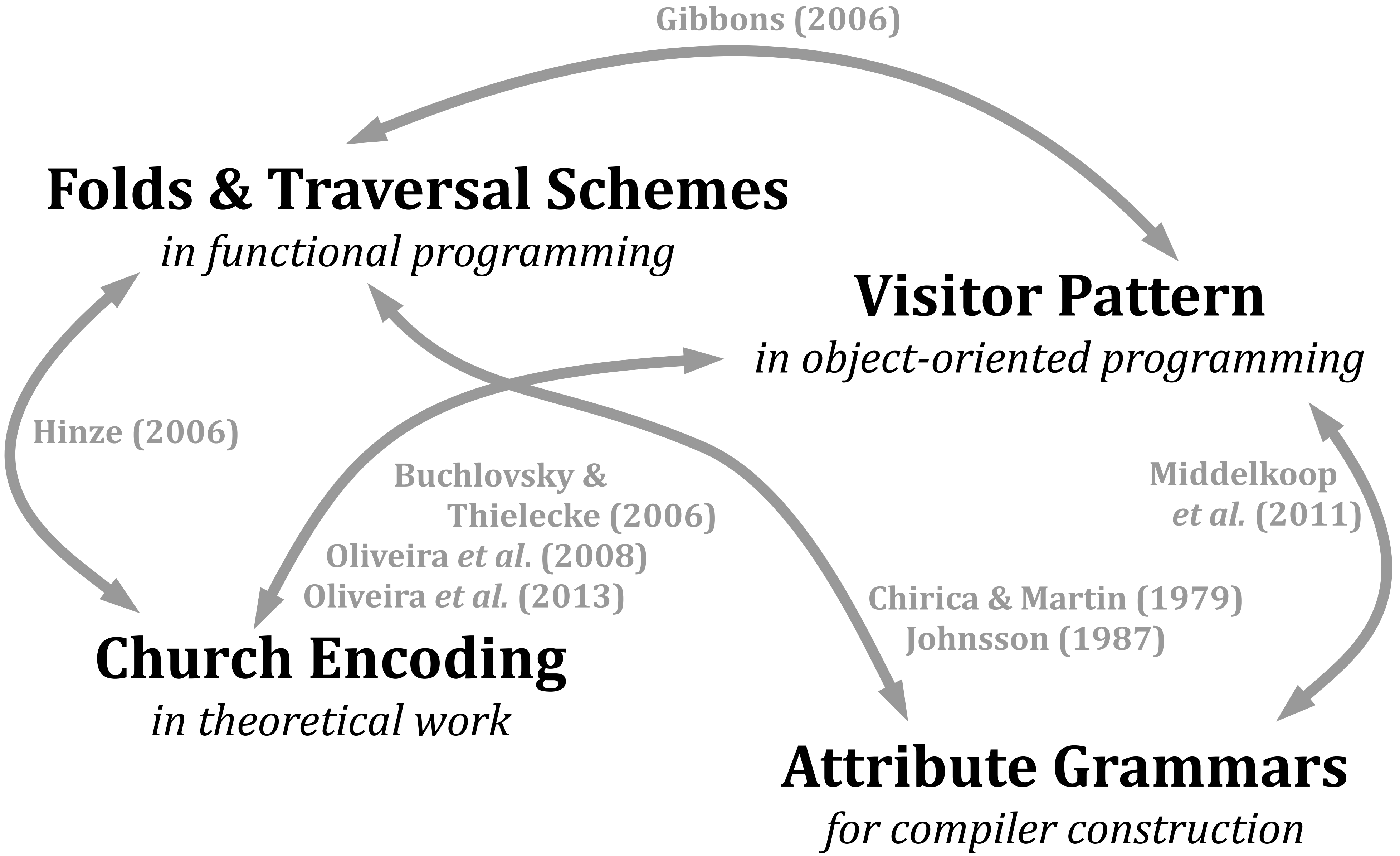
in object-oriented programming

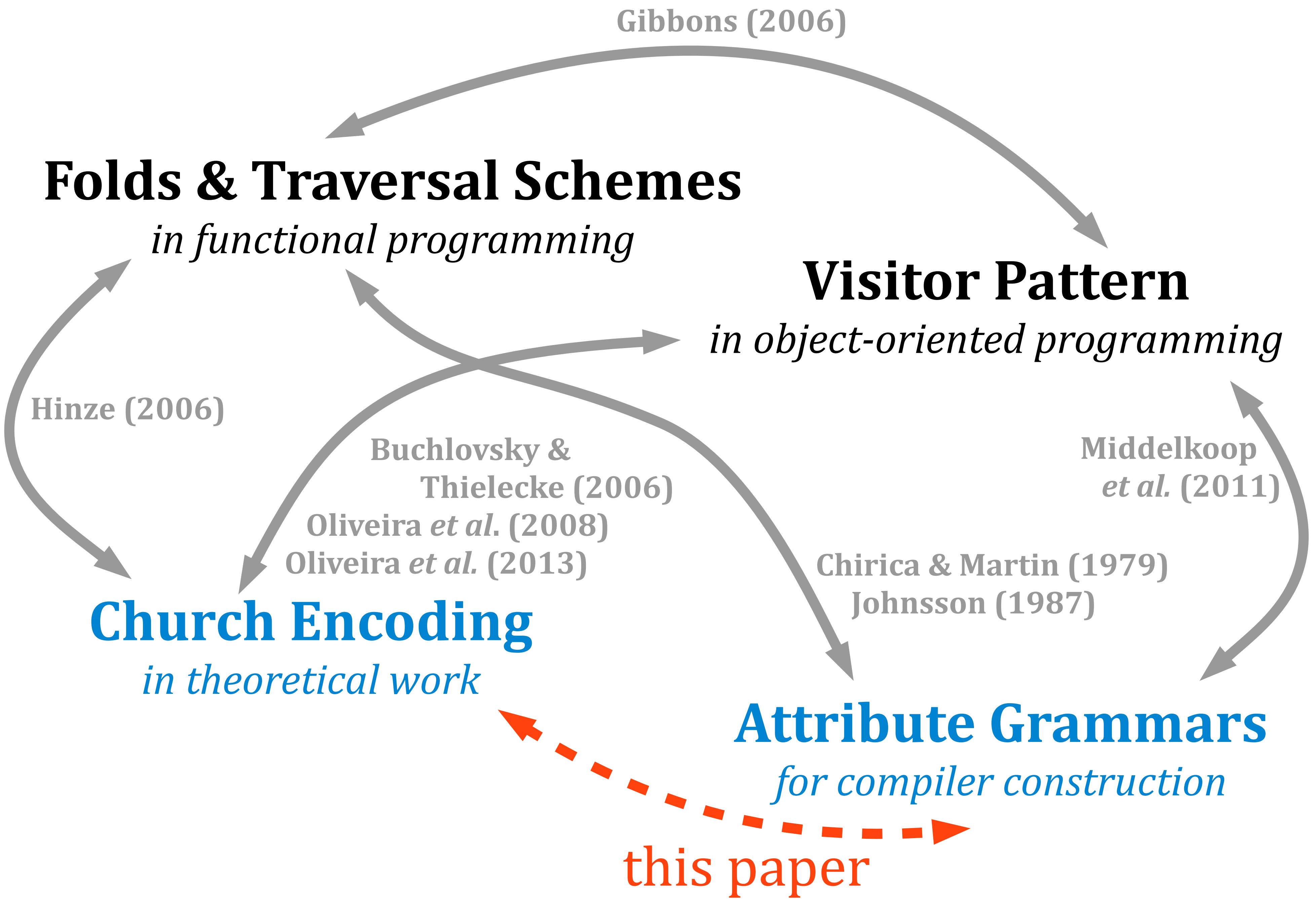
Church Encoding

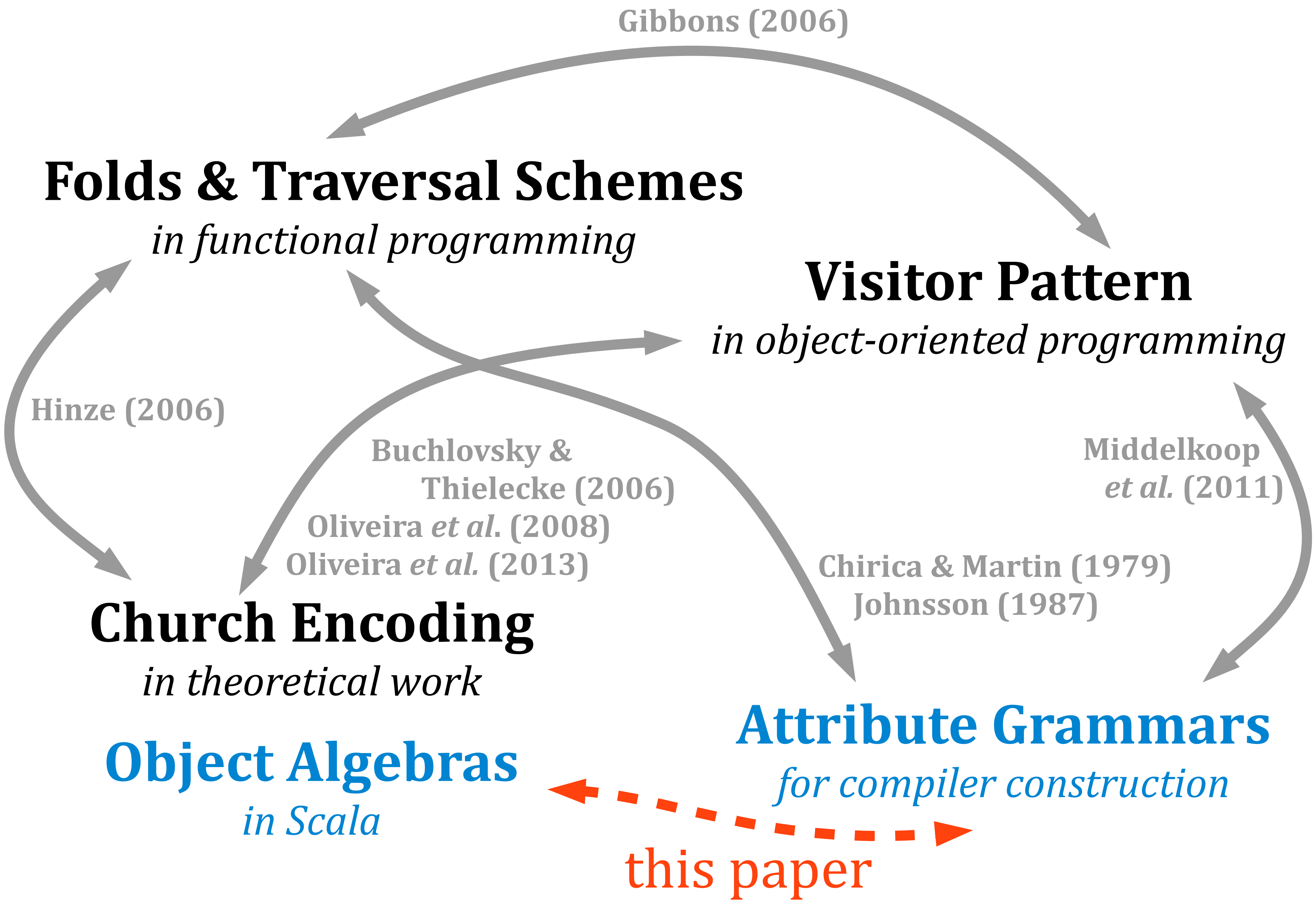
in theoretical work

Attribute Grammars

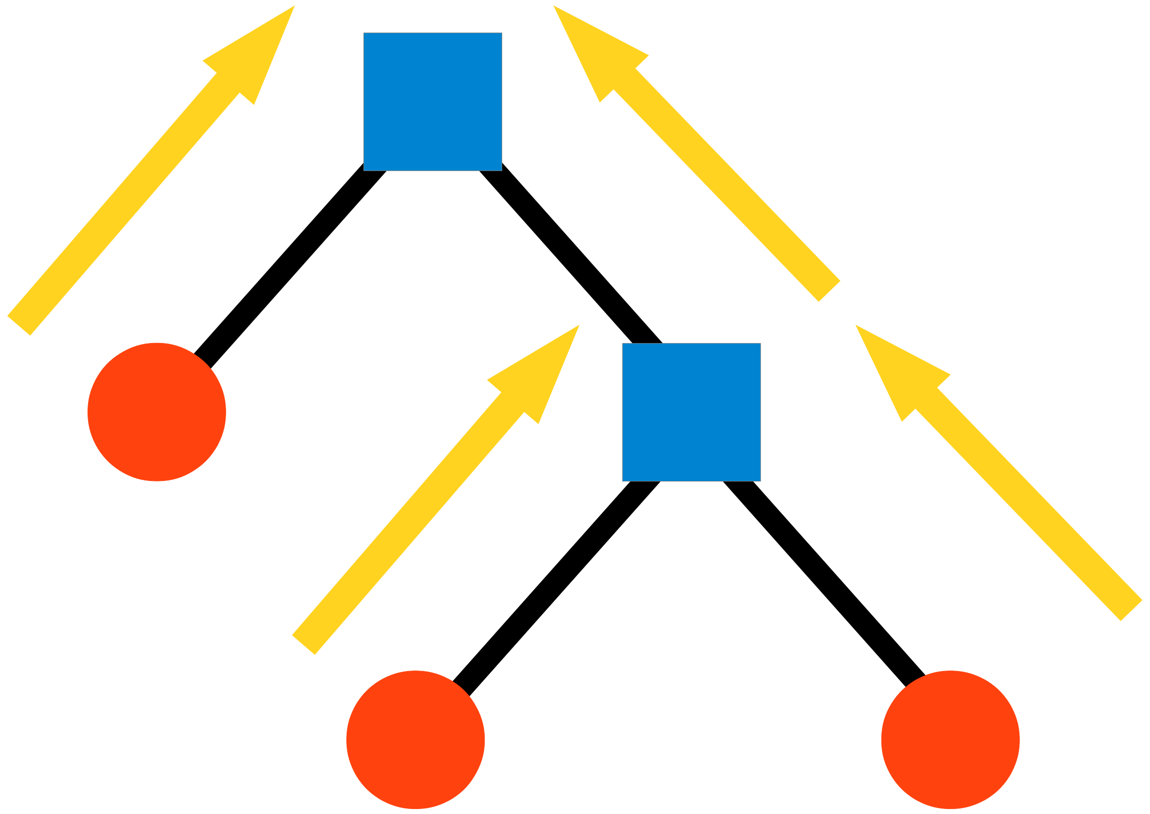
for compiler construction


Gibbons (2006)
Folds & Traversal Schemes
in functional programming
Visitor Pattern
in object-oriented programming
Hinze (2006)
Buchlovsky & Thielecke (2006)
Oliveira et al. (2008)
Oliveira et al. (2013)
Middelkoop et al. (2011)
Chirica & Martin (1979)
Johnsson (1987)
Church Encoding
in theoretical work
Attribute Grammars
for compiler construction


Gibbons (2006)
Folds & Traversal Schemes
in functional programming
Visitor Pattern
in object-oriented programming
Hinze (2006)
Buchlovsky & Thielecke (2006)
Oliveira et al. (2008)
Oliveira et al. (2013)
Middelkoop et al. (2011)
Chirica & Martin (1979)
Johnsson (1987)
Church Encoding
in theoretical work
Attribute Grammars
for compiler construction
this paper


Gibbons (2006)
Folds & Traversal Schemes
in functional programming
Visitor Pattern
in object-oriented programming
Hinze (2006)
Buchlovsky & Thielecke (2006)
Oliveira *et al.* (2008)
Oliveira *et al.* (2013)
Middelkoop *et al.* (2011)
Chirica & Martin (1979)
Johnsson (1987)
Church Encoding
in theoretical work
Attribute Grammars
for compiler construction
Object Algebras
in Scala
this paper

Bottom-Up Data Flow



Synthesized Attributes

Grammar

$e_0 \;\to\; n$ { Lit }

$e_1 \;\to\; e_2$ "+" 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

$e_0 \;\rightarrow\; n \qquad\qquad\qquad \{\text{ Lit }\}$

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

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
}
```

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

$e_0 \to n$ { Lit }

$e_1 \to e_2$ "+" 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

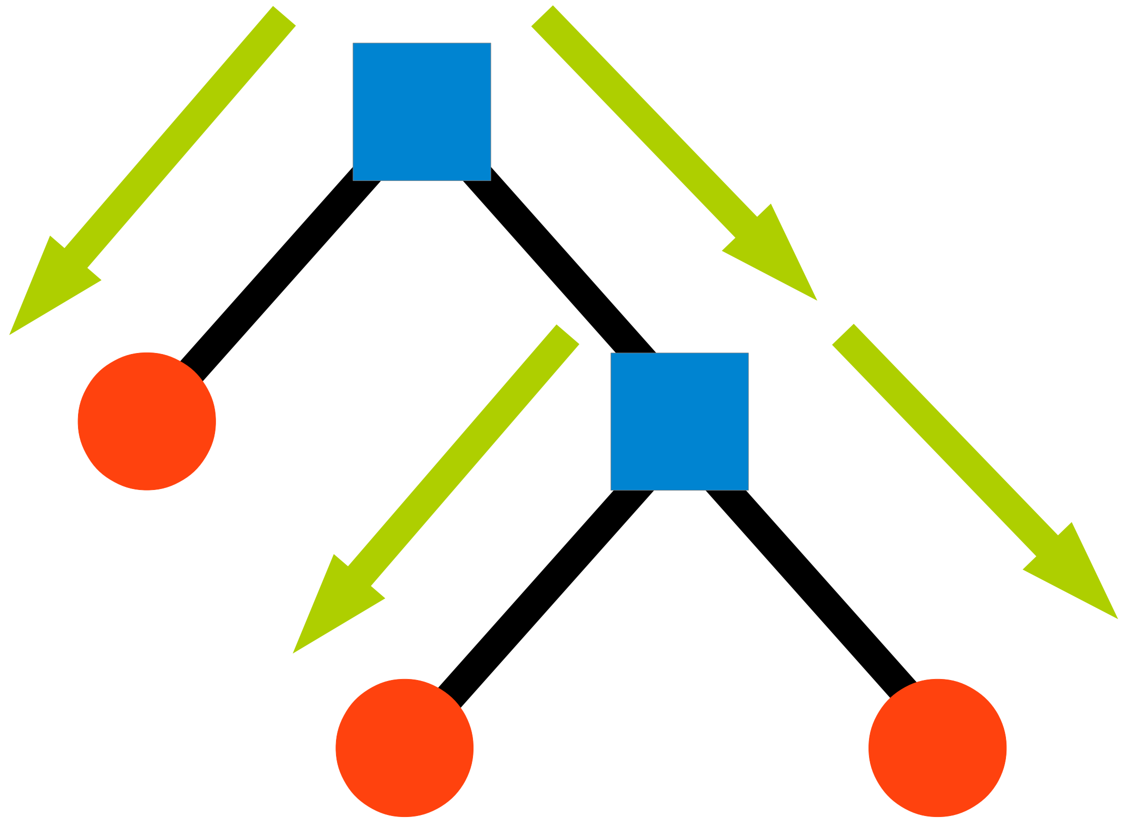
Signature

```
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 Add_1: $E \Rightarrow E$
   def Add_2: $(E, E) \Rightarrow E$
}

Algebra

val Alg = **new** Sig[Bool] {
   def Add_1 = $e \Rightarrow$ **true**
   def Add_2 = $(e_1, e_2) \Rightarrow$ **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 }

Equations

e_2.left = **true**

e_3.left = **false**

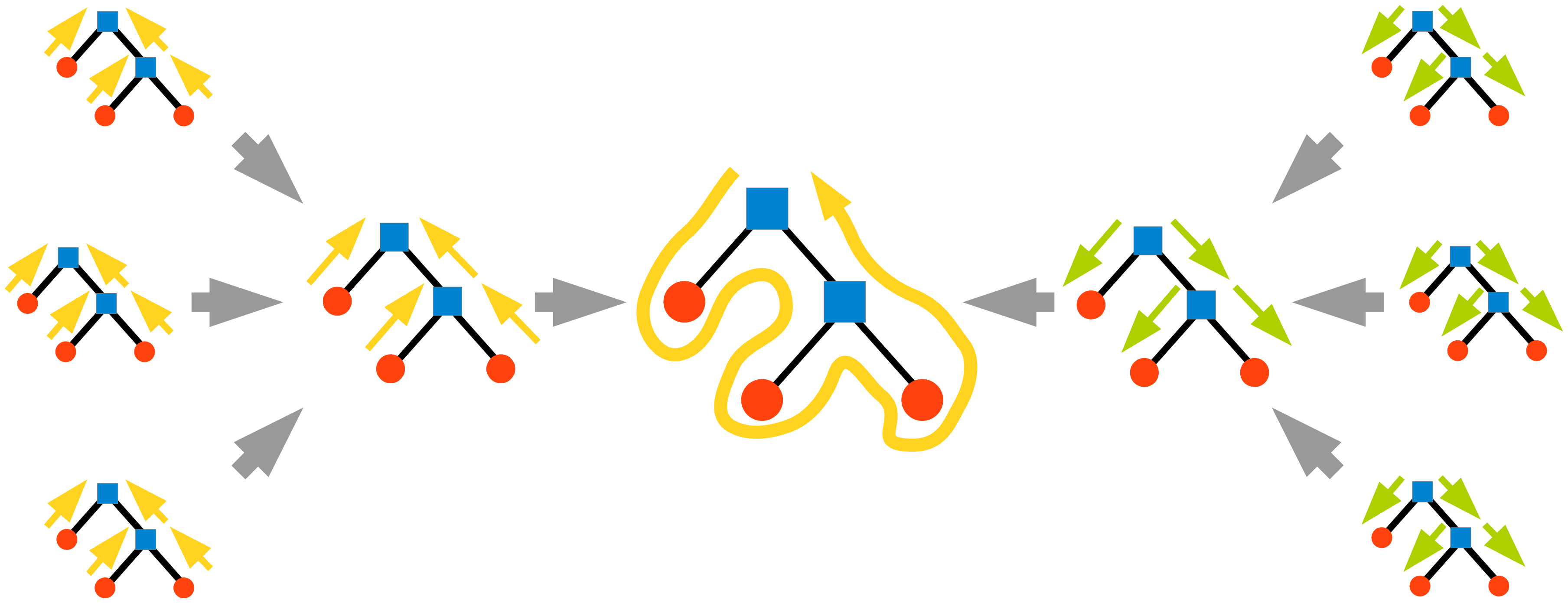
Signature

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

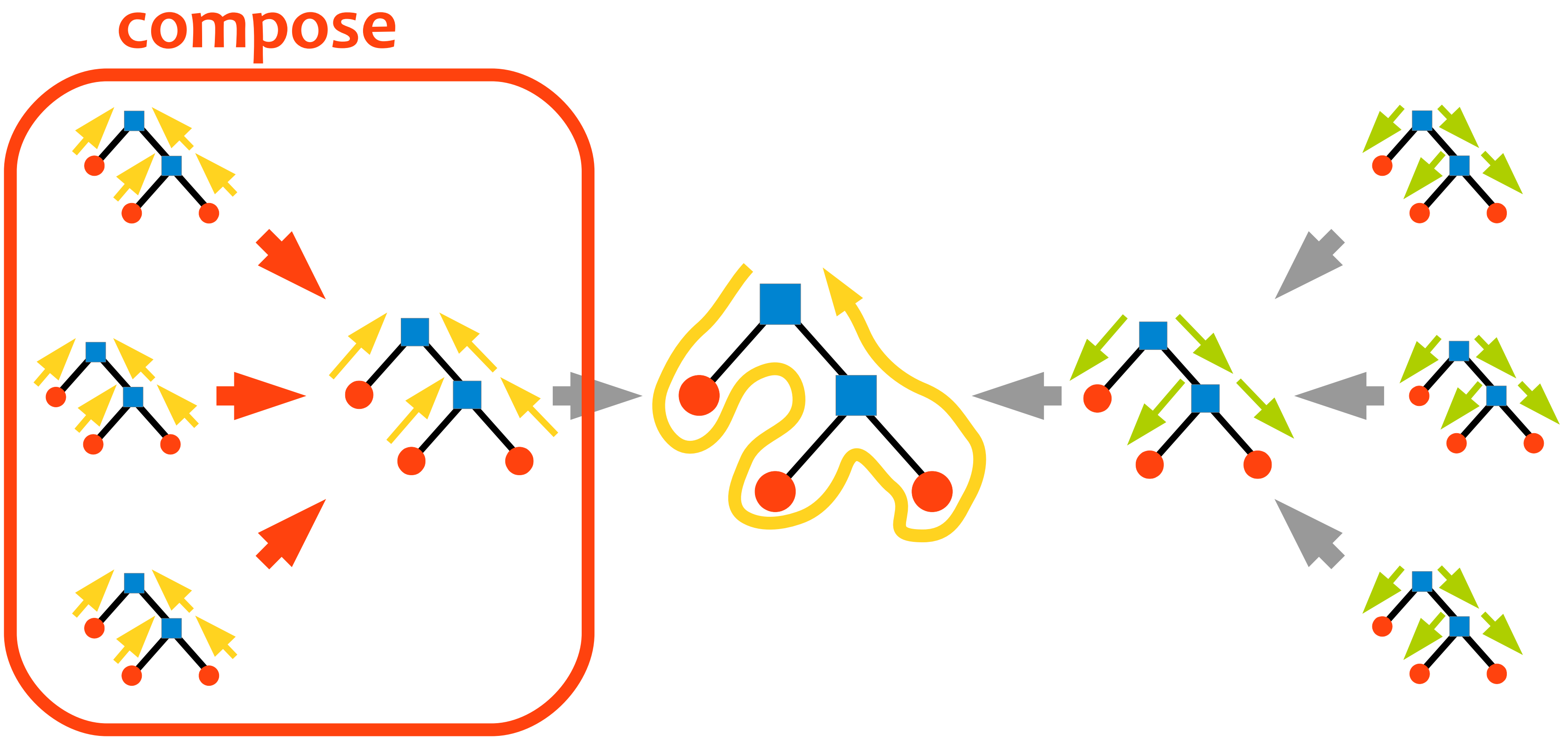
Algebra

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

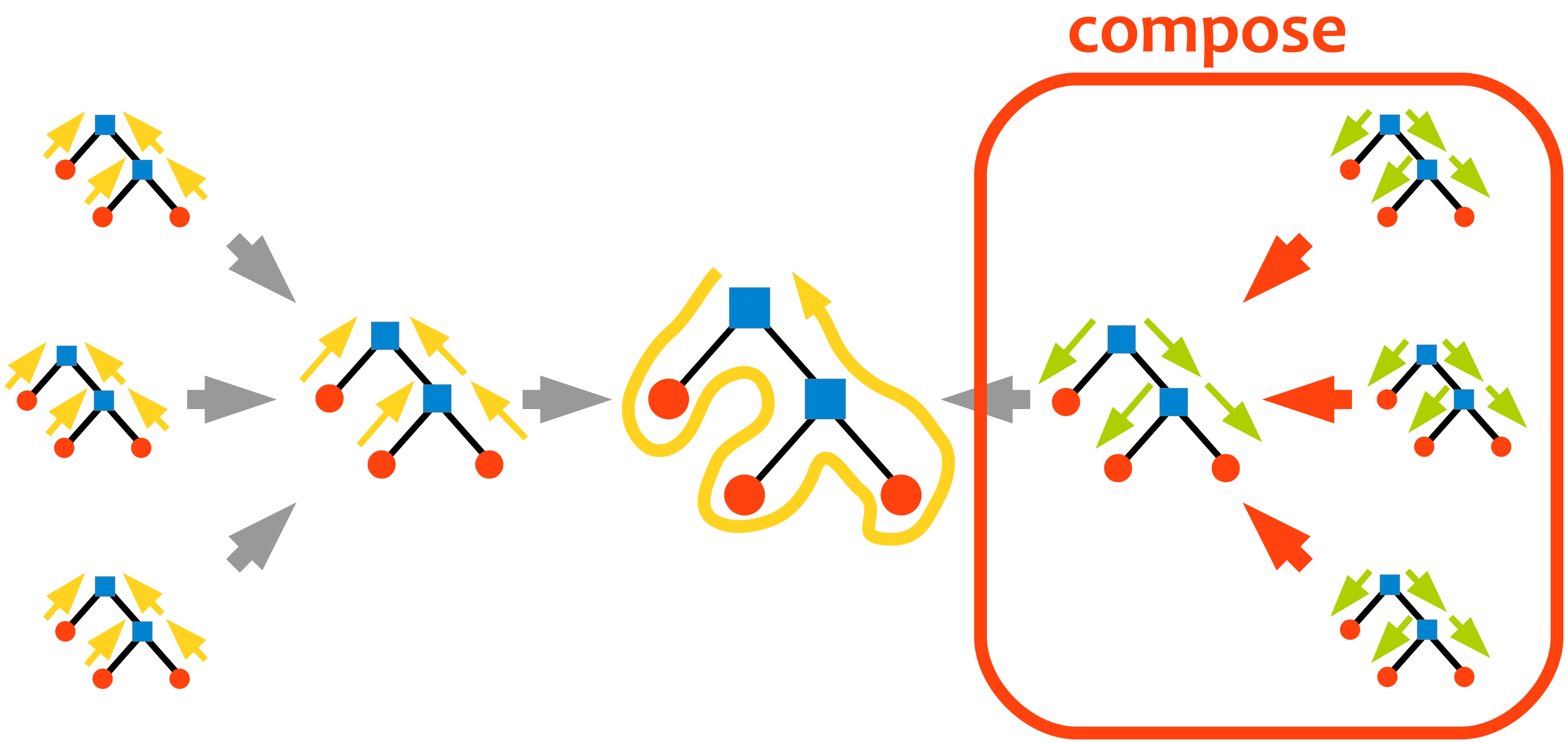
Composition



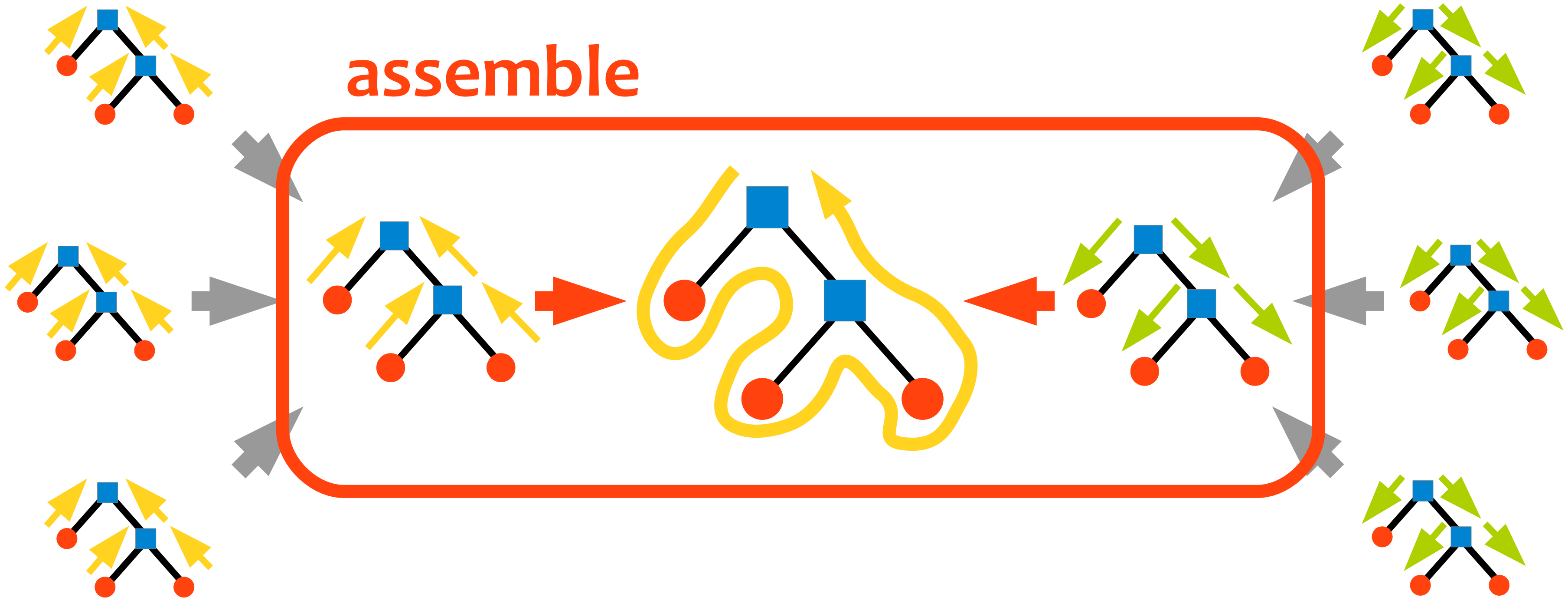
Composition

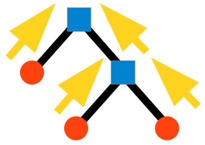
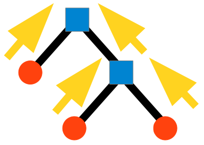
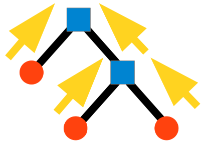


Composition



Composition



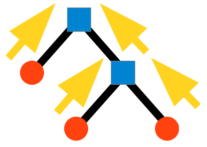
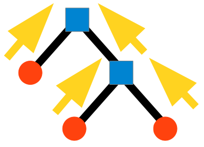
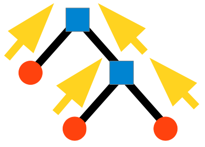
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

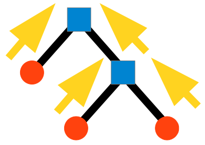
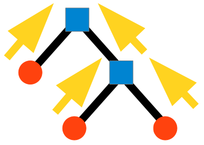
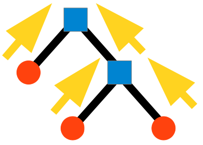
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

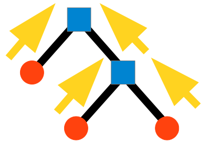
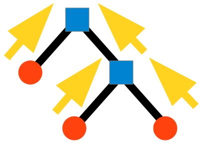
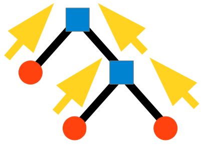
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

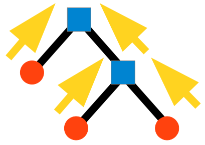
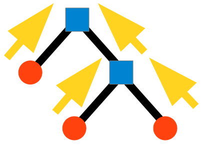
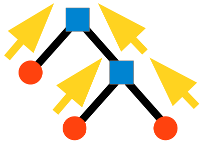
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

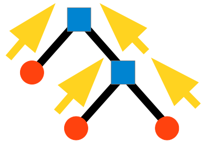
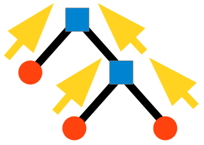
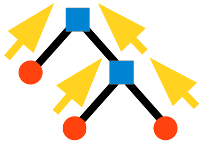
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

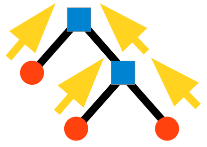
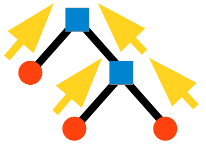
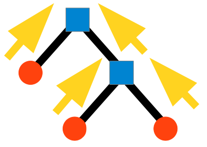
compose( , ) = 

Extensible Records

trait HasValue {**def** value: Int}
trait HasLeft {**def** left: Bool}
def mix[*A*, *B*]: (*A*, *B*) ⇒ $\boxed{A \textbf{ with } B}$

Dependency Tracking

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

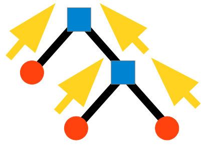
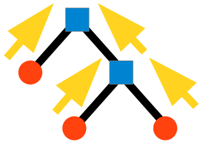
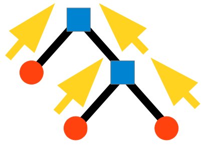
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

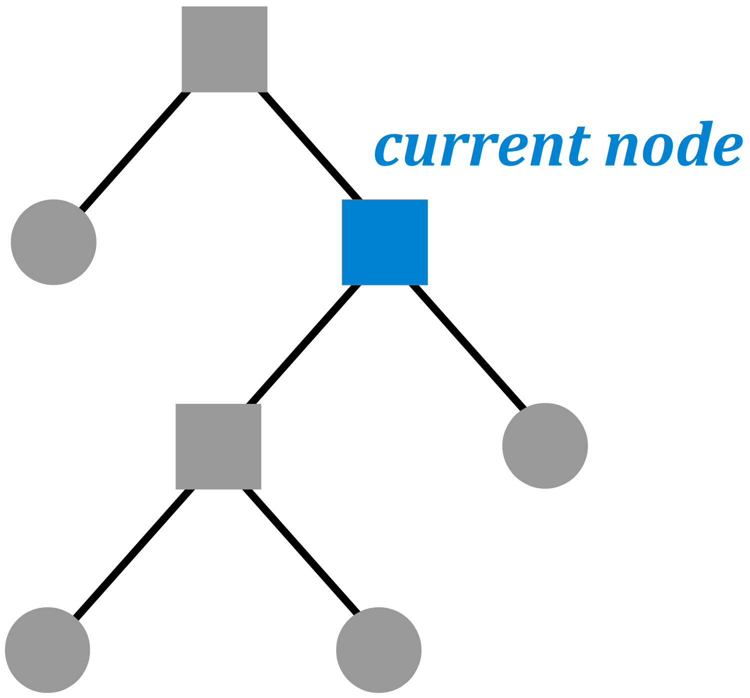
compose( , ) = 

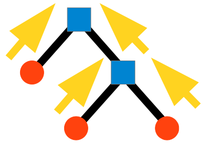
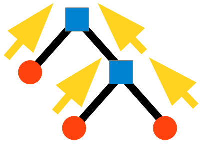
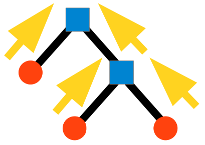
Extensible Records

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

Dependency Tracking

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



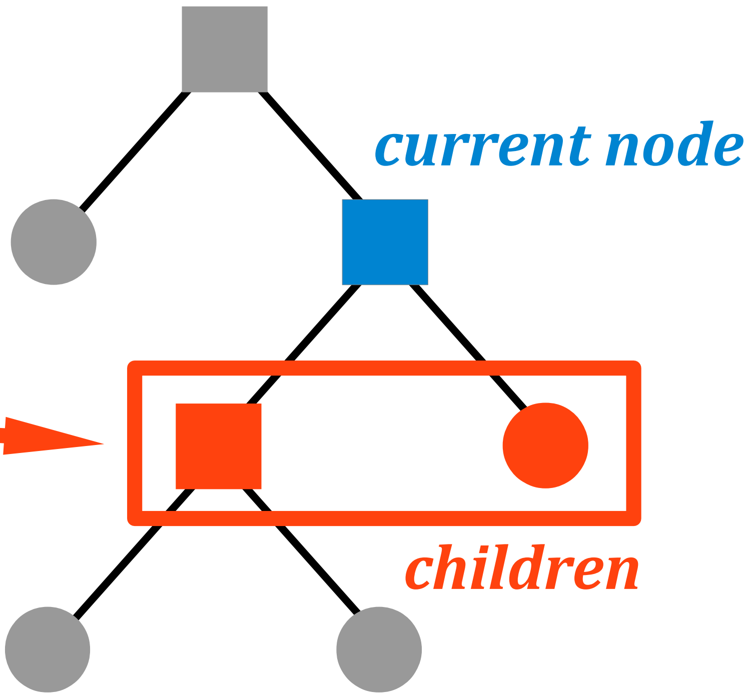
compose( , ) = 

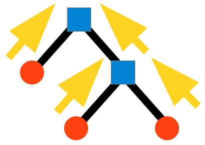
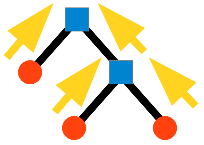
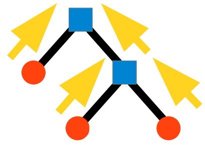
Extensible Records

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

Dependency Tracking

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



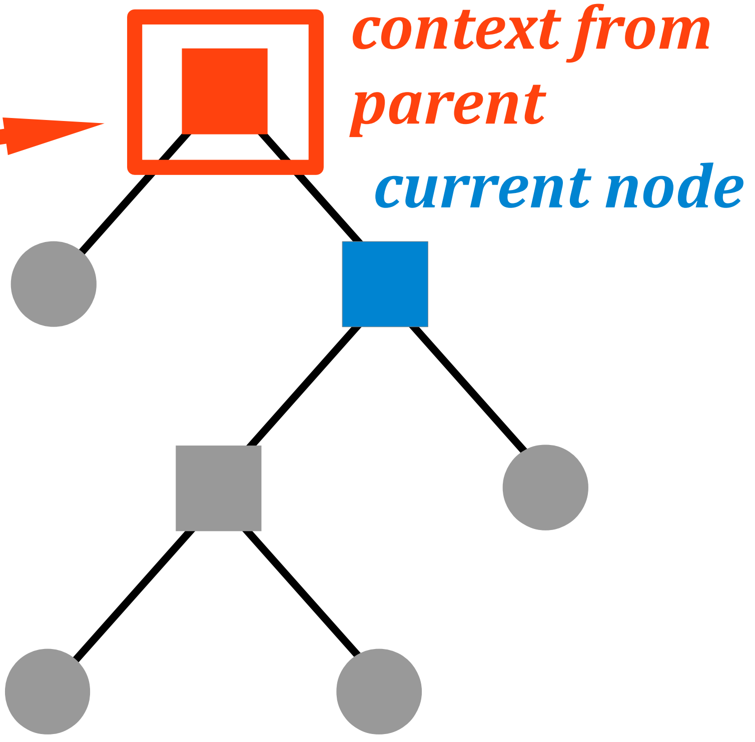
compose( , ) = 

Extensible Records

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

Dependency Tracking

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



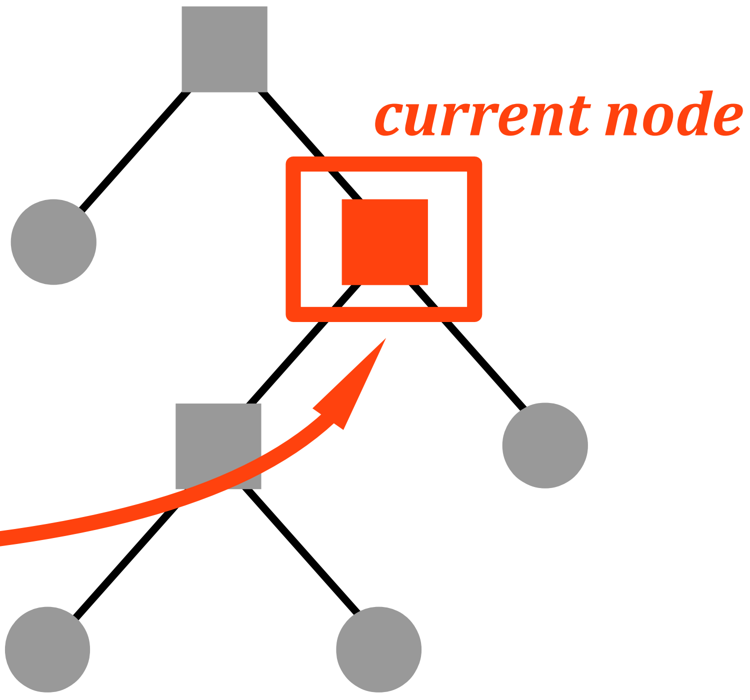
compose(,) =

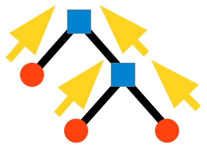
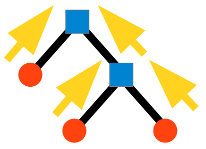
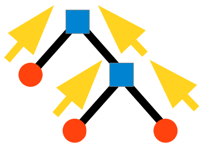
Extensible Records

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

Dependency Tracking

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



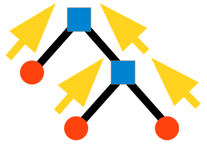
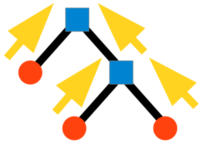
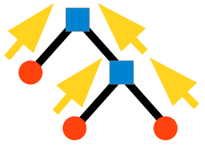
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

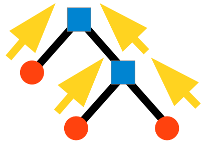
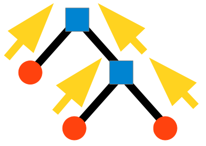
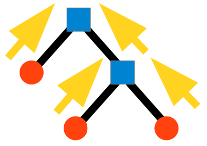
compose( , ) = 

Extensible Records

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

Dependency Tracking

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

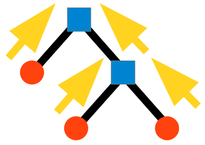
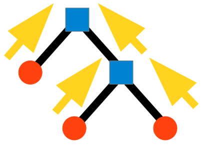
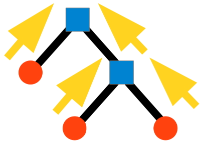
compose( , ) = 

Extensible Records

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

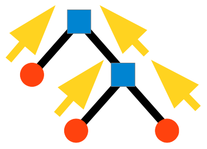
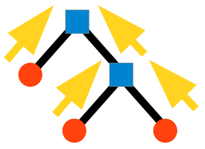
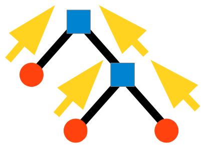
Dependency Tracking

trait Sig[-*E*, -*C*, +*O*] {
    def Lit: Int ⇒ *C* ⇒ *O*
    def $\boxed{\text{Add: } (E, E) \Rightarrow C \Rightarrow O}$
}

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

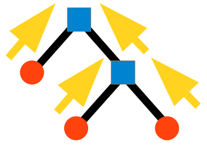
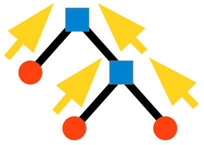
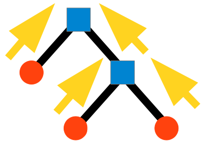
def compose

$[E_1, C_1, O_1, E_2, C_2 >: C_1$ **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

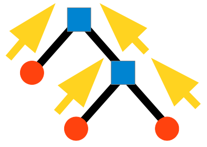
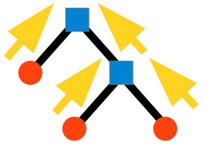
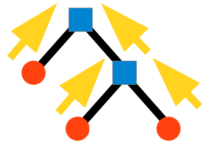
def compose

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

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

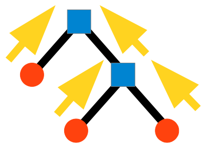
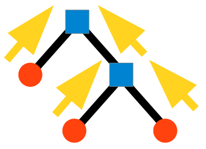
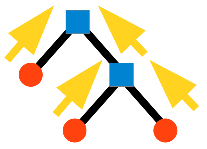
$alg_2: Sig[E_2, C_2, O_2]):$

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

compose( , ) = 

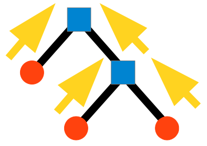
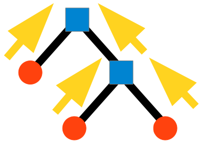
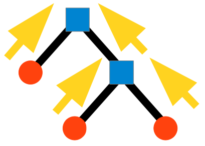
Composing two algebras

```
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
        [$E_1$, $C_1$, $O_1$, $E_2$, $C_2$ >: $C_1$ 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

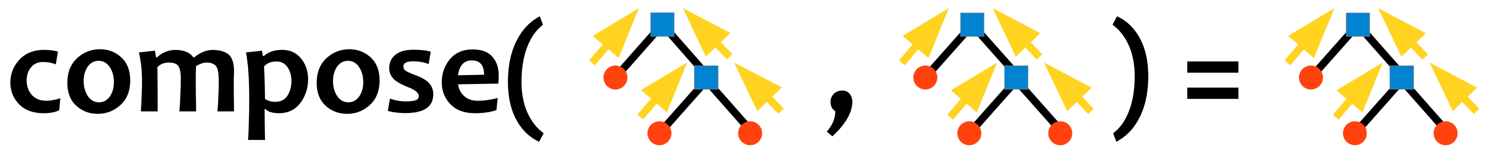
def compose

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

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

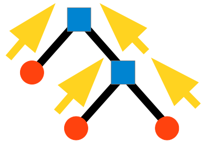
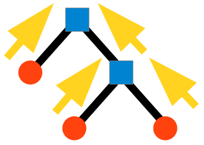
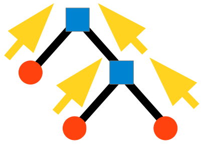
$alg_2: Sig[E_2, C_2, O_2])$:

Sig$[\boxed{E_1 \textbf{ with } E_2}, C_1, O_1$ **with** $O_2]$



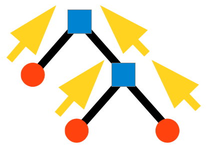
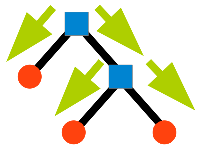
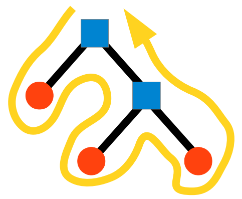
Composing two algebras

```
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 $E_2$, $\boxed{C_1}$ $O_1$ with $O_2$]
```

compose( , ) = 

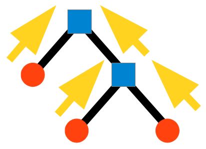
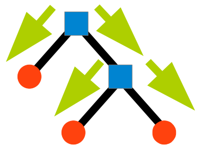
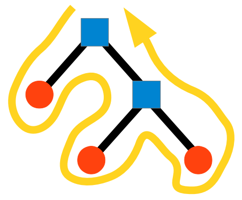
Composing two algebras

def compose
    $[E_1, C_1, O_1, E_2, C_2 >: C_1$ **with** $O_1, O_2]$
    $(alg_1: \text{Sig}[E_1, C_1, O_1],$
    $alg_2: \text{Sig}[E_2, C_2, O_2]):$
        $\text{Sig}[E_1$ **with** $E_2, C_1, \boxed{O_1 \textbf{ with } O_2}]$

assemble( , ) = 

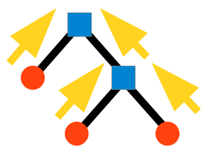
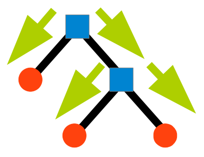
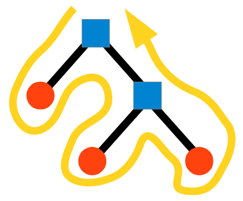
Assembling a one-pass traversal

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

assemble( , ) = 

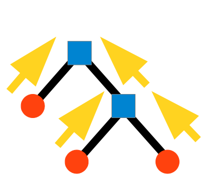
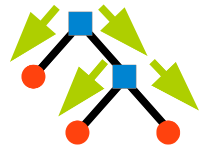
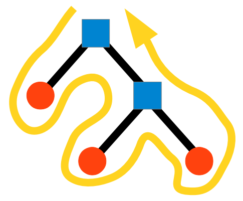
Assembling a one-pass traversal

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

assemble( , ) = 

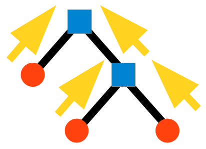
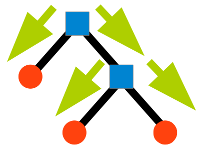
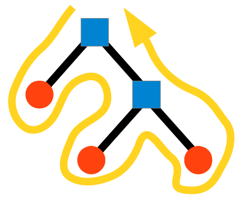
Assembling a one-pass traversal

```
def  assemble
        [C, O]
        (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

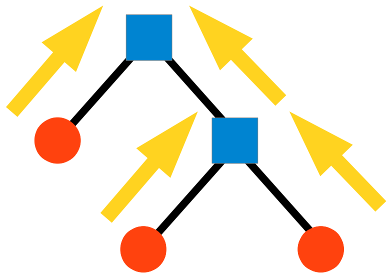
def assemble
    [C, O]
    (alg$_1$: Sig$_1$[C with O, C, O],
     alg$_2$: $\boxed{\text{Sig}_2\text{[C with O, C, C]}}$):
       Sig[C ⇒ C with O]

assemble( **,**  **) =** 

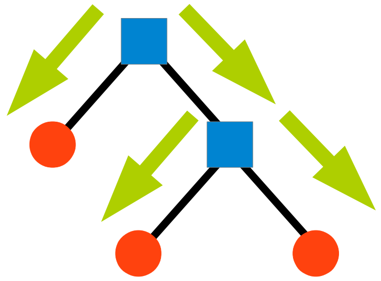
Assembling a one-pass traversal

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

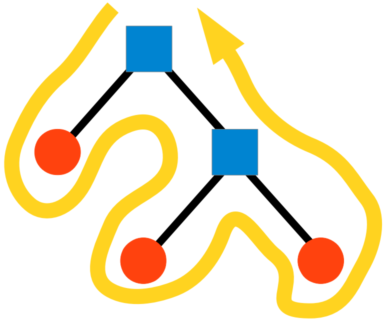
Results



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

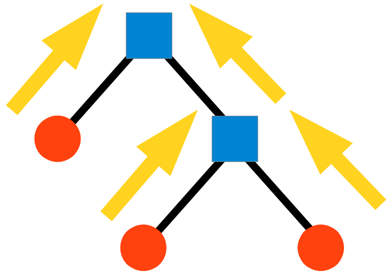


We **extend** object algebras
to support **inherited attributes**
(top-down data flow)

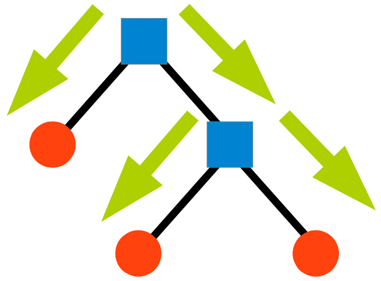


We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

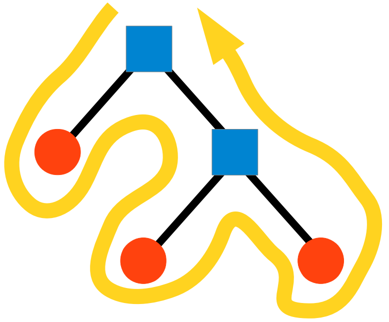
Results



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

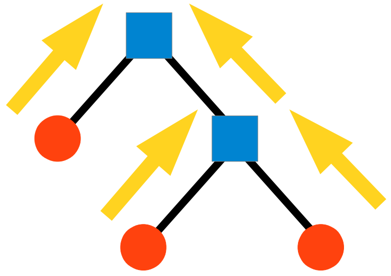


We **extend** object algebras
to support **inherited attributes**
(top-down data flow)

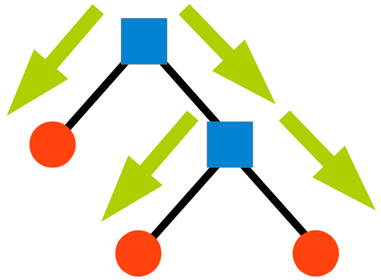


We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

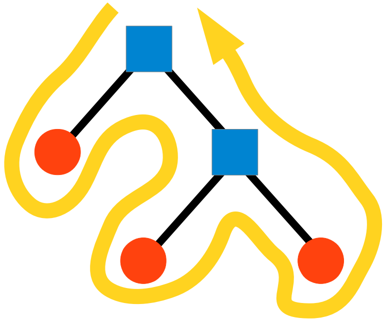
Results



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

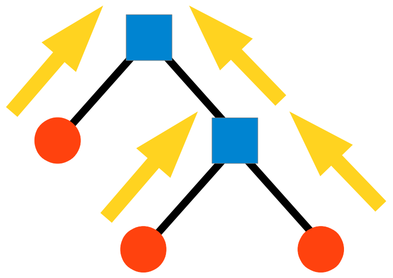


We **extend** object algebras
to support **inherited attributes**
(top-down data flow)

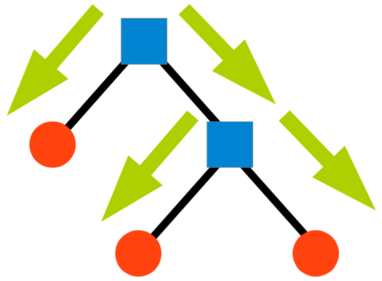


We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

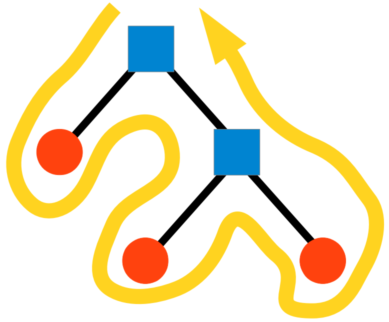
Results



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



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



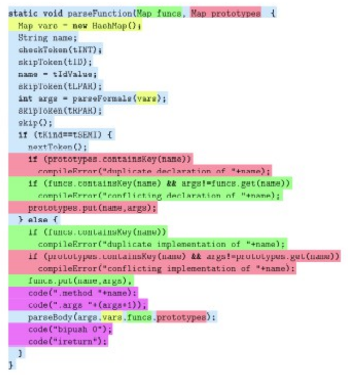
We **assemble** multiple algebras
to support **L-attributed grammars**
(arbitrary one-pass compiler)

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 of the present paper)

Monolithic compiler

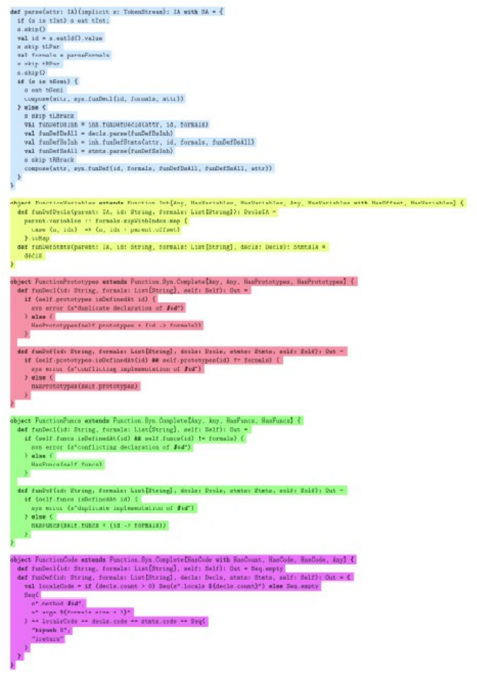
1 file
807 lines of Java code
entangled



```
static void parseFunction(Map funcs, Map prototypes  {
  Map vars = new HashMap();
  String name;
  checkToken(tINT);
  skipToken(tID);
  name = tIdValue;
  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, syn.funDecl(id, formals, attr))
  } else {
    s skip tLBrack
    val funDefDsInh = inh.funDefDecls(attr, id, formals)
    val funDefDsAll = decls.parse(funDefDsInh)
    val funDefSsInh = inh.funDefStmts(attr, id, formals, funDefDsAll)
    val funDefSsAll = stmts.parse(funDefSsInh)
    s skip tRBrack
    compose(attr, syn.funDef(id, formals, funDefDsAll, funDefSsAll, attr))
  }
}

object FunctionVariables extends Function.Inh[Any, HasVariables, HasVariables, Any, HasVariables with HasOffset, HasVariables] {
  def funDefDecls(parent: IA, id: String, formals: List[String]): DeclsIA =
    parent.variables ++ formals.zipWithIndex.map {
      case (n, idx)  => (n, idx + parent.offset)
    }.toMap
  def funDefStmts(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.Complete[HasCode 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.

Conclusions

$$\begin{tikzcd}[column sep=huge] \text{\textbf{\textit{Object Algebras}} \textit{in Scala}} \arrow[r, leftrightarrow, dashed, bend right, "\text{Rendel et al. (2014)}"'] & \text{\textbf{Attribute Grammars} \textit{for compiler construction}} \end{tikzcd}$$

Conclusions

Object Algebras
in Scala

Attribute Grammars
for compiler construction

$$\begin{tikzcd}\text{Object Algebras} \arrow[r, leftrightarrow, dashed, bend right, "\text{Rendel et al. (2014)}"'] & \text{Attribute Grammars}\end{tikzcd}$$

Benefits for OA

- Support for
  inherited attributes
- Access to
  extensive AG research
- Future work:
  encode more AG features

Conclusions

Object Algebras
in Scala

Attribute Grammars
for compiler construction

Rendel et al. (2014)

Benefits for OA

- Support for inherited attributes
- Access to extensive AG research
- Future work: encode more AG features

Benefits for AG

- Modular, scalable, and compositional encoding
- Embedding enables abstraction via meta language
- Future work: AG compiler to object algebras

Conclusions

Object Algebras
in Scala

Attribute Grammars
for compiler construction

Rendel et al. (2014)

Benefits for OA

- Support for inherited attributes
- Access to extensive AG research
- Future work: encode more AG features

Benefits for AG

- Modular, scalable, and compositional encoding
- Embedding enables abstraction via meta language
- Future work: AG compiler to object algebras

Thank You!