Functional Programming, Object-Oriented Programming and Algebras!

Bruno Oliveira (bruno@cs.hku.hk)
The University of Hong Kong

10th ACM SIGPLAN Workshop on Generic Programming (WGP 2014)

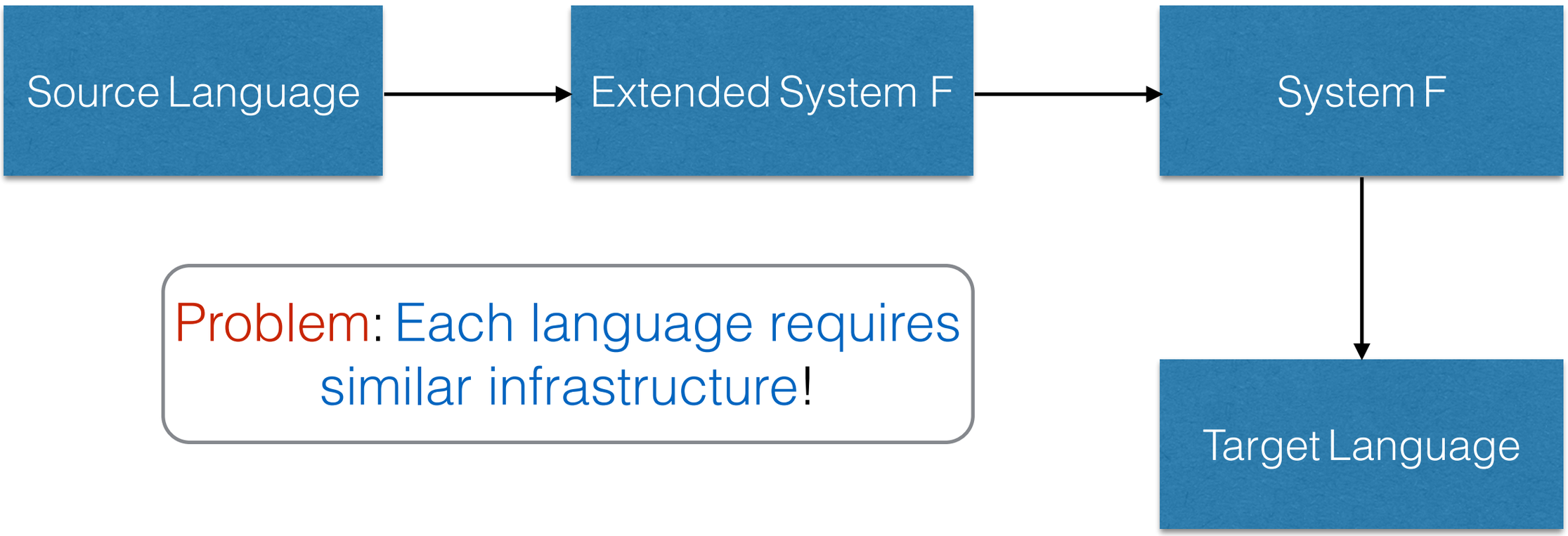
Long-term Goal of My Research

How to achieve modularity, type-safety and reuse?

(Without requiring a PhD in Type-Theory!)

Motivation

- A compiler for a (functional) language typically requires a number of languages and transformations between these languages:



Language 1

$$\textbf{data}\ Exp_1 = Num_1\ Int \mid Add_1\ Exp_1\ Exp_1 \mid Minus_1\ Exp_1\ Exp_1$$

$$\begin{aligned}
&eval_1 :: Exp_1 \rightarrow Int \\
&eval_1\ (Num_1\ x) &&= x \\
&eval_1\ (Add_1\ e1\ e2) &&= eval_1\ e1 + eval_1\ e2 \\
&eval_1\ (Minus_1\ e1\ e2) &&= eval_1\ e1 - eval_1\ e2
\end{aligned}$$

Repeated code!

Language 2

$$\textbf{data}\ Exp_2 = Num_2\ Int \mid Add_2\ Exp_2\ Exp_2 \mid Minus_2\ Exp_2\ Exp_2 \mid Neg_2\ Exp_2$$

$$\begin{aligned}
&eval_2 :: Exp_2 \rightarrow Int \\
&eval_2\ (Num_2\ x) &&= x \\
&eval_2\ (Add_2\ e1\ e2) &&= eval_2\ e1 + eval_2\ e2 \\
&eval_2\ (Minus_2\ e1\ e2) &&= eval_2\ e1 - eval_2\ e2 \\
&eval_2\ (Neg_2\ e) &&= -\ (eval_2\ e)
\end{aligned}$$

Repeated code!

$$\begin{aligned}
&narrow_{21} :: Exp_2 \rightarrow Exp_1 \\
&narrow_{21}\ (Num_2\ x) &&= Num_1\ x \\
&narrow_{21}\ (Add_2\ e1\ e2) &&= Add_1\ (narrow_{21}\ e1)\ (narrow_{21}\ e2) \\
&narrow_{21}\ (Minus_2\ e1\ e2) &&= Minus_1\ (narrow_{21}\ e1)\ (narrow_{21}\ e2) \\
&narrow_{21}\ (Neg_2\ e) &&= Minus_1\ (Num_1\ 0)\ (narrow_{21}\ e)
\end{aligned}$$

Algebras

- Algebras: A recurring abstraction that shows up in both FP & OOP.
- In recent years Algebras proved useful to solve various extensibility/modularity issues in FP & OOP.
- Goal 1: Overview of existing work; lessons learned.
- Goal 2: Promote algebras as a good alternative to algebraic datatypes/OO hierarchies.
- Goal 3: Vision for future work: with adequate language support, algebras can provide a simple solution to our modularity problem.

Algebras??

- Algebra is a very broad term.
- What is meant by Algebras in this talk?
  - Essentially (variants of) F-Algebras

F-Algebras???

- F-Algebras have been known and used for a long time in Functional Programming:
  - F-Algebras show up in the definition of recursion patterns such as catamorphisms/folds.
- Widely used in the Algebra of Programming ... lots of work in the end of 80's and 90's

Meijer et al., Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire (1991)

Bird and de Moor, The Algebra of Programming (1997)

… and many, many others!

F(old)-Algebras

```
data List = Nil | Cons Int List
```

```
foldList :: (Int -> a -> a) -> a -> List -> a
foldList f k Nil           = k
foldList f k (Cons x xs)   = f x (foldList f k xs)

type FoldAlgebra a = (Int -> a -> a, a)
```

F(old)-Algebras

```
data Exp = Lit Int | Add Exp Exp

type FoldAlgebraExp a = (Int -> a, a -> a -> a)

foldExp :: FoldAlgebraExp a -> Exp -> a
foldExp (l, a) (Lit x)          = l x
foldExp (l, a) (Add e1 e2)  =
   a (foldExp (l, a) e1) (foldExp (l, a) e2)
```

F-Algebras

- F-Algebras are isomorphic to Fold-Algebras; use sums-of-products instead of products-of-functions

F a -> a

```
type FAlgebraExp a = Either Int (a,a) -> a

cataExp :: FAlgebraExp a -> Exp -> a
cataExp f (Lit x)          = f (Left x)
cataExp f (Add e1 e2)  =
   f (Right (cataExp f e1, cataExp f e2))
```

Varieties of Algebras

- Lesson 1: F-Algebras show up in various (isomorphic) forms

- FoldAlgebra: (Int -> a, a -> a -> a)

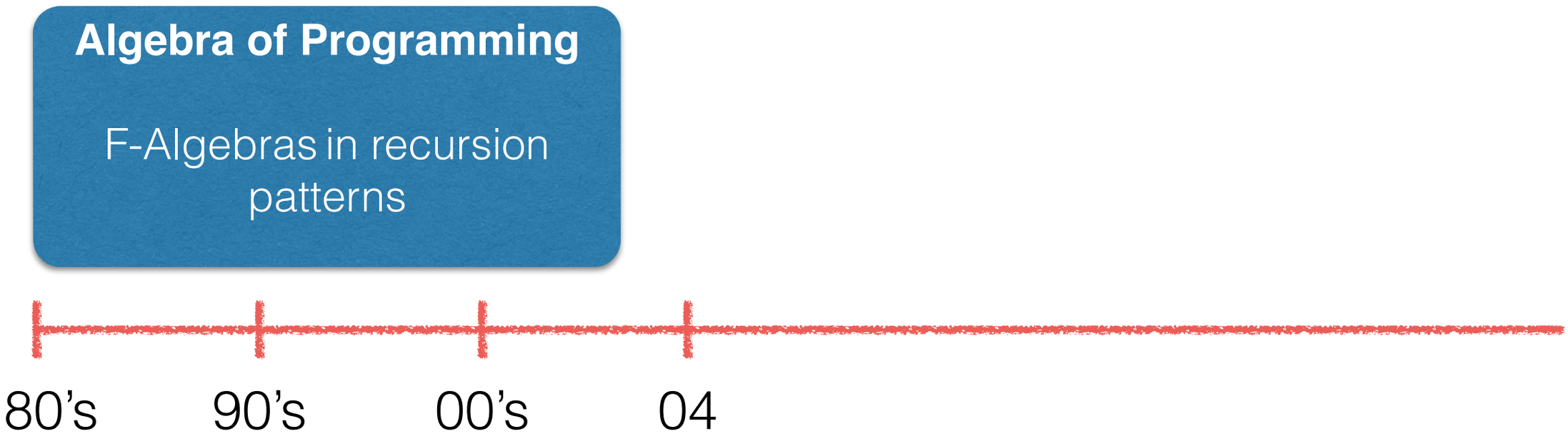
  Int -> a x a -> a -> a

- F-Algebra: Either Int (a,a) -> a

  Int + a x a

Timeline

Timeline of Algebras-related research



Church Encodings

Church Encodings

- Between 2003-2008 a few things were happening in two areas:
  - Haskell/FP community:
    - The community discovers GADTs and their applications
  - Type-Theory/OO:
    - A type-theoretic explanation of the VISITOR pattern shows up.

Encodings of GADTs

- Researchers working in datatype-generic programming (DGP) were trying to create lightweight approaches (library-based instead of language-based)

- Type representations such as:

```
data Rep t where
   RInt     :: Rep Int
   RProd :: Rep a -> Rep b -> Rep (a,b)
```

Turned out to be quite handy!

Cheney and Hinze, A Lightweight Implementation of Generics and Dynamics, HW (2002)
Hinze, Fun with phantom types, The Fun of Programming (2003)

Encodings of GADTs

- Problem: Back in 2003 there were no GADTs in Haskell: they had to be encoded somehow!

- One proposal, due to Hinze, was to encode GADTs using type classes:

```
class Generic rep where
  rint      :: rep Int
  rprod     :: rep a -> rep b -> rep (a,b)
```

A Fold Algebra!

```
class Repr t where
  repr :: Generic rep => rep t
```

A Fold!

Hinze, Generics for the Masses, ICFP (2004)

Encodings of GADTs

Folding representations:

```
foldRep :: rep Int ->
           (forall a b . rep a -> rep b -> rep (a,b)) ->
           Rep t -> rep t
foldRep rint rprod RInt          = rint
foldRep rint rprod (RProd r1 r2) =
    rprod (foldRep rint rprod r1) (foldRep rint rprod r2)
```

Encodings of GADTs

- The essence of this encoding were type-theoretic encodings of datatypes (in particular Church encodings)

- Type classes are just a clever way to employ Haskell features to model encodings of GADTs.

- Church encodings can be viewed as a way to model algebraic datatypes using folds.

Hinze, Generics for the Masses, JFP (2006)
Oliveira and Gibbons, TypeCase: A Design Pattern for Type-Indexed Functions, HW (2005)

Type-Theoretic Visitors

- Around the same timeframe, Buchlovsky and Thielecke, were also investigating uses of Church Encodings:

```
interface Visitor<β> {
  β visitLeaf(Int x);
  β visitNode(β x, β y);
}
```

A Fold Algebra!

```
interface BinTree {
  <α> α accept(Visitor<α> v);
}
```

A Fold!

Buchlovsky and Thielecke, A Type-theoretic Reconstruction of the Visitor Pattern, MFPS (2005)

Type-Theoretic Visitors

- The essence of the type-theoretic explanation was also based on type-theoretic encodings of datatypes.

- The use of generic interfaces is just the way to model encodings of ADTs in OO languages.

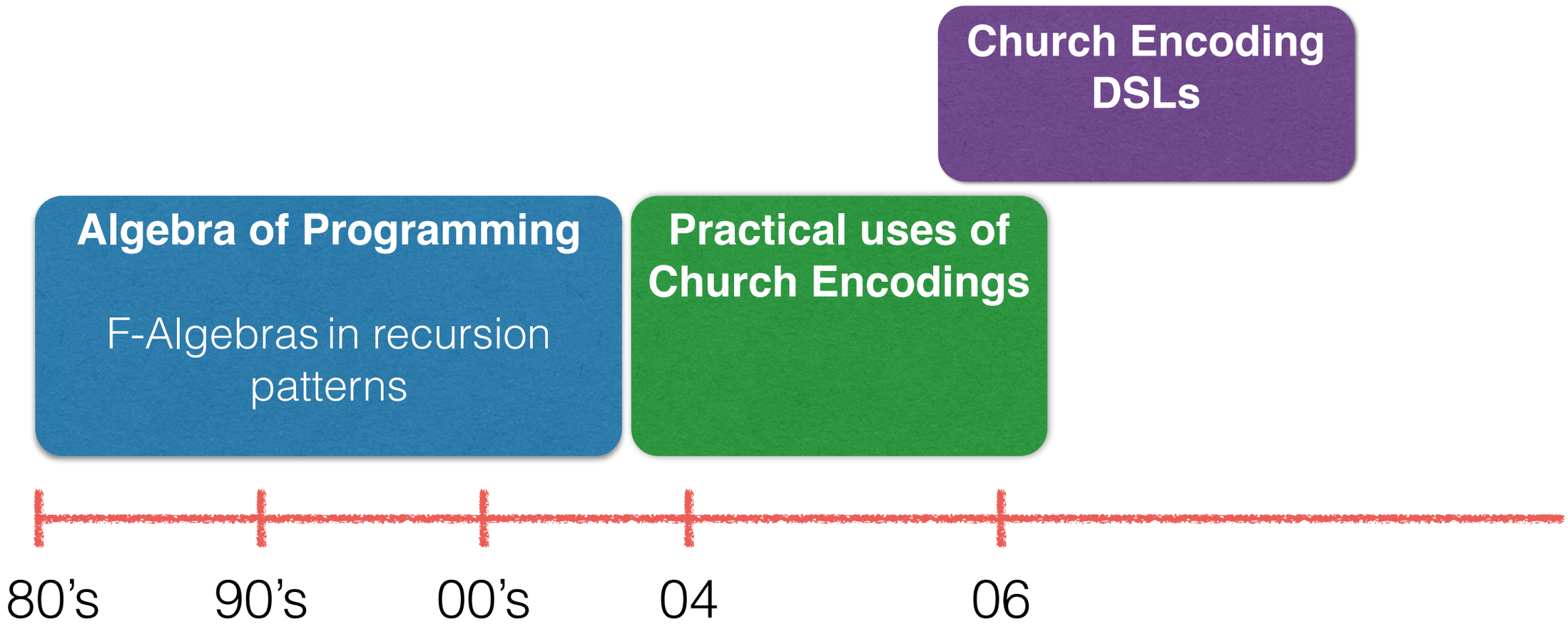
Applications

- Lesson 2: Algebras are useful to model tree structures in many languages/paradigms, without requiring algebraic datatypes.
- In OO languages we can use generic interfaces to model algebras.
- In Haskell there are various ways to model algebras: type classes being one of them.
- One (now very popular!) application is embedded DSLs: using encodings of datatypes allowed something between shallow and deep embeddings.

Carrete et al., Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages, APLAS (2007) & JFP (2009)
Hofer et al., Polymorphic Embedding of DSLs, GPCE (2008)
Rompf and Odersky, Lightweight Modular Staging, GPCE (2010)

Timeline

Timeline of Algebras-related research



Extensibility and Modularity

The Expression Problem

- The Expression Problem is a well-know modularity problem.

- After 2006 many solutions to the expression problem using algebras were found/popularized.

Wadler, The Expression Problem, Java Genericity Mailing list (1998)

The Expression Problem

- How do you add multiplication to the following type of expressions modularly?

```
data Exp = Lit Int | Add Exp Exp
```

Closed/Not Extensible

```
type FoldAlgebraExp a = (Int -> a, a -> a -> a)

foldExp :: FoldAlgebraExp a -> Exp -> a
foldExp (l, a) (Lit x)          = l x
foldExp (l, a) (Add e1 e2)  =
   a (foldExp (l, a) e1) (foldExp (l, a) e2)
```

Extensible Encodings

- The expression problem affects Datatype-Generic Programming libraries.

- Hinze's original Generics for the masses approach did not allow extensible generic functions.

- Oliveira et al. noted the connection between the problems and suggested a solution.

- Oliveira et al., Extensible and Modular Generics for the Masses, TFP (2006)

Extensible Encodings

- With some modifications, the Generics for the Masses approach can be made extensible.

```
class Generic rep where
  rint      :: rep Int
  rprod     :: rep a -> rep b -> rep (a,b)
```

Extensibility!

```
class Generic rep => GenericExt rep where
  rplus     :: rep a -> rep b -> rep (a,b)
```

Extensible Encodings

- In this solution algebras are encoded as type classes.

- Extended algebras can be defined using subclasses.

Datatypes a la Carte

- A different (and now very popular) solution to the Expression Problem was introduced in 2008 by Swierstra

- The solution relies on F-Algebras

Swierstra, Datatypes a la Carte, JFP (2008)

Datatypes a la Carte

```
data NumF a = Num Int deriving Functor
data AddF a = Plus a a deriving Functor

class Eval f where
  eval :: f Int -> Int
instance Eval NumF where
  eval (Num x) = x
instance Eval AddF where
  eval (Plus e1 e2) = e1 + e2

data Fix f = In {out :: f (Fix f)}

fold :: Functor f => (f a -> a) -> Fix f -> a
fold alg = alg . fmap (fold alg) . out
```

F-Algebra!

Fold!

Datatypes a la Carte

```
data MulF a = Mul a a deriving Functor

class Eval f where
  eval :: f Int -> Int
instance Eval NumF where
  eval (Num x) = x
instance Eval AddF where
  eval (Plus e1 e2) = e1 + e2
instance Eval MulF where
  eval (Mul e1 e2) = e1 * e2
```

Extensible Visitors/ Object Algebras

- In OO languages we can also solve the Expression Problem with Algebras.

- Interface inheritance can be used to achieve extensibility.

Oliveira, Modular Visitor Components, ECOOP (2009)
Oliveira and Cook, Extensibility for the Masses, ECOOP (2012)

Extensible Visitors/ Object Algebras

```
interface IntAlg<A> {
  A lit(int x);
  A add(A e1, A e2);
}

interface IntBoolAlg<A> extends IntAlg<A> {
  A bool(Boolean b);
  A iff(A e1, A e2, A e3);
}
```

Fold Algebra!

Extensibility!

Abstracting Away

Two different (but isomorphic) solutions using Algebras.

Fold-Algebra: Int -> a x a -> a -> a x a -> a -> a

Extensibility!

F-Algebra: Int + a x a + a x a -> a

Extensibility

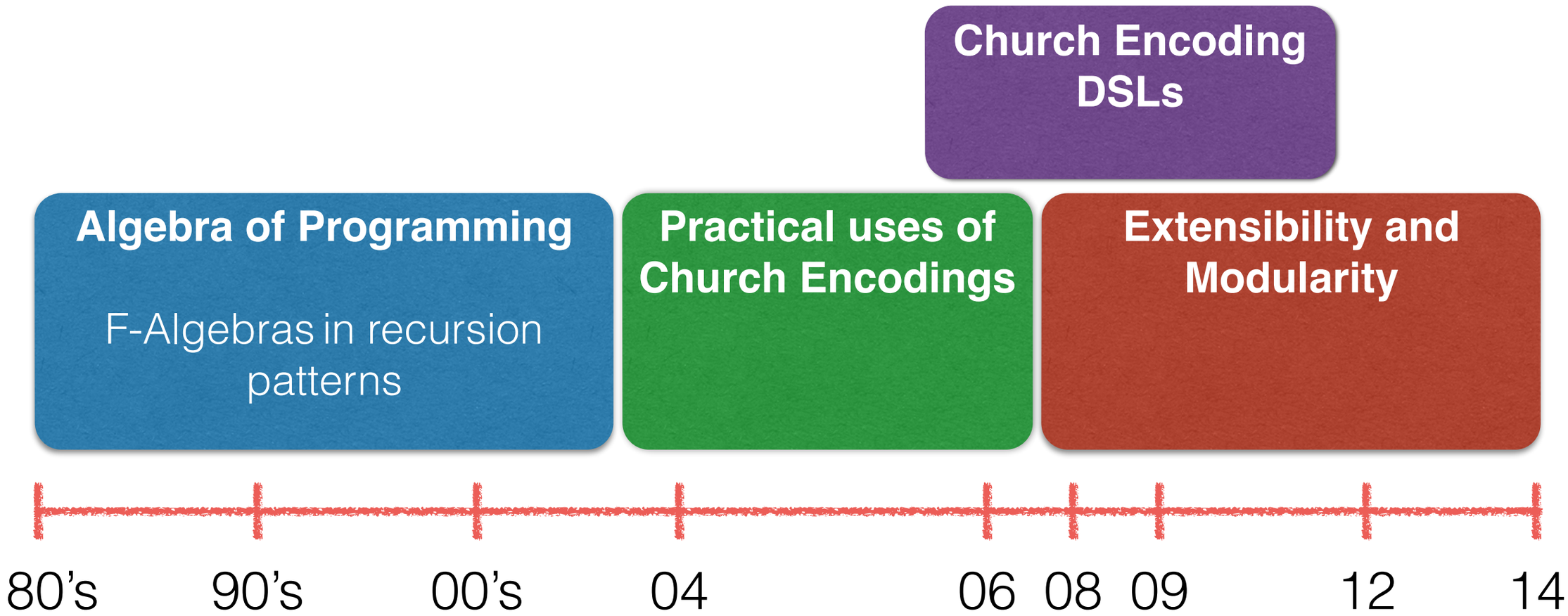
- Lesson 3: Algebras support extensibility/modularity.

- With algebraic datatypes, adding new cases modularly is not possible!

- With OO hierarchies adding new methods modularly to an existing interface is not possible!

Timeline

Timeline of Algebras-related research



More Work

- There is too much work on algebras and modularity to summarize here:
  - Increasing the expressive power of algebras:
    - Oliveira et al., Feature-Oriented Programming with Object Algebras, ECOOP (2013)
    - Rendel et al., From Object Algebras to Attribute Grammars, OOPSLA (2014)
  - Ornamental Algebras (a different dimension of modularity)
    - McBride, Ornamental Algebras, Algebraic Ornaments, JFP (2010)
  - Modular Reasoning
    - Delaware et al., Meta-Theory a la Carte, POPL (2013)
    - Delaware et al., Modular Monadic Meta-Theory, ICFP (2013)

Lessons Learned

- Lesson 1: F-Algebras show up in various (isomorphic) forms

- Lesson 2: Algebras are useful to model tree structures in many languages/paradigms.

- Lesson 3: Algebras support extensibility/modularity.

- **Lesson 4**: Current solutions using algebras still require heavy encodings.

The Future: Programming Language Support?

Why PL Support for Algebras?

- Currently algebras have to be encoded with other language constructs.

- Often algebras need to be composed. Some of the composition operators have to be encoded.

- Algebras (and generalizations) use parametric polymorphism/generics. In more complex algebras there is too much parametrization going on.

Conclusions

- Existing solutions using various encodings of algebras have been extremely useful to better understand modularity issues.

- I think we should continue exploring such encodings to improve our understanding.

- However, I believe we also need to look at improving PL support to make solutions practical.

Thank you!

Language support?

```
algebra interface ExpAlg where
    sort Exp
    Lit     : Int -> Exp
    Add  : Exp -> Exp -> Exp

interface IEval where
    eval : Int

algebras Eval implements ExpAlg where
    type Exp = IEval

    eval@(Lit x)           = x
    eval@(Add e1 e2)  = e1.eval + e2.eval
```