香 港 大 學
THE UNIVERSITY OF HONG KONG

EVF:
An Extensible and Expressive Visitor Framework for Programming Language Reuse

Weixin Zhang

Joint work with Bruno C. d. S. Oliveira

June 23, 2017

Motivation

Motivation

- New PLs/DSLs are needed and existing PLs are evolving all the time

Motivation

- New PLs/DSLs are needed and existing PLs are evolving all the time
- However, creating and maintaining a PL is hard
  - syntax, semantics, tools …
  - implementation effort
  - expert knowledge

Motivation

- New PLs/DSLs are needed and existing PLs are evolving all the time
- However, creating and maintaining a PL is hard
  - syntax, semantics, tools …
  - implementation effort
  - expert knowledge
- PLs share a lot of features
  - variable declaration, arithmetic operations …

Motivation

- New PLs/DSLs are needed and existing PLs are evolving all the time
- However, creating and maintaining a PL is hard
  - syntax, semantics, tools …
  - implementation effort
  - expert knowledge
- PLs share a lot of features
  - variable declaration, arithmetic operations …
- But it is hard to materialize *conceptual reuse* into *software engineering reuse*

Language Components

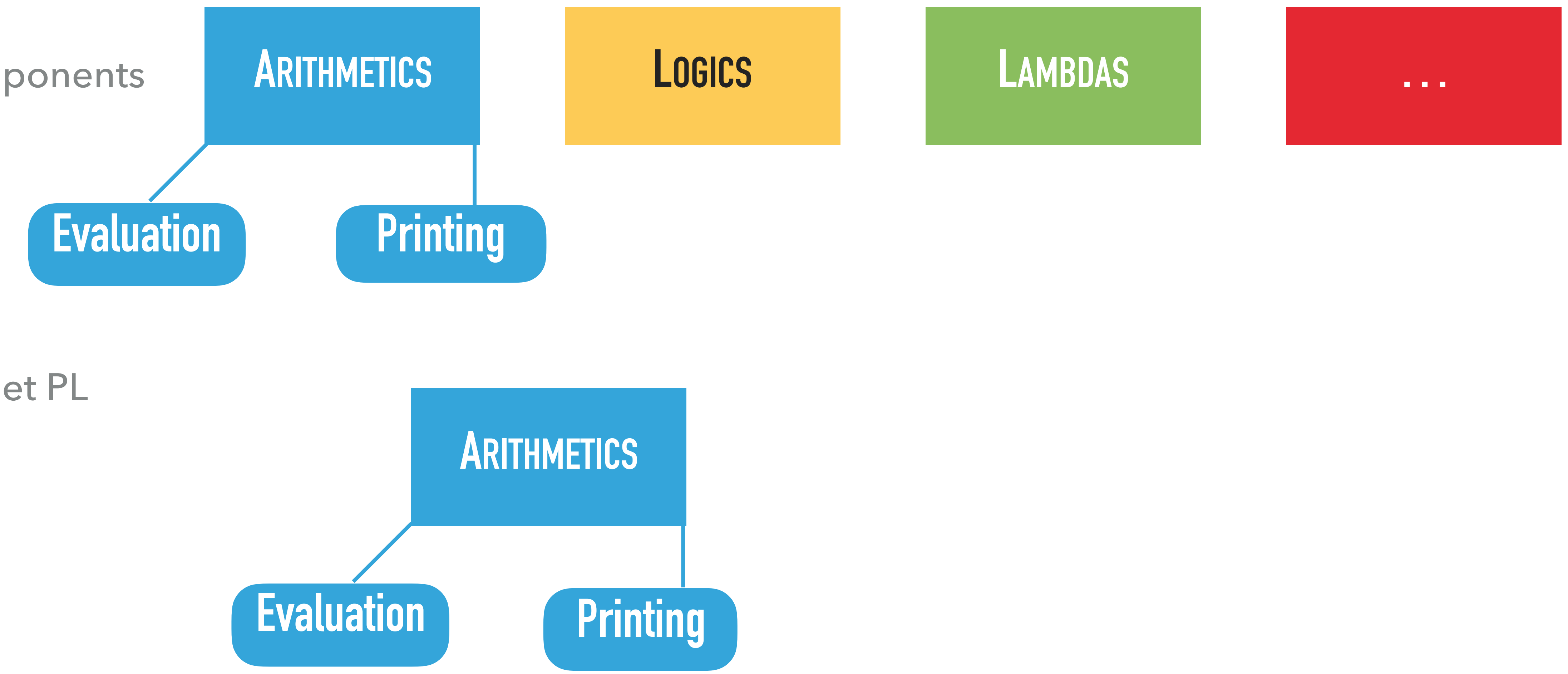
Language Components

Language Components

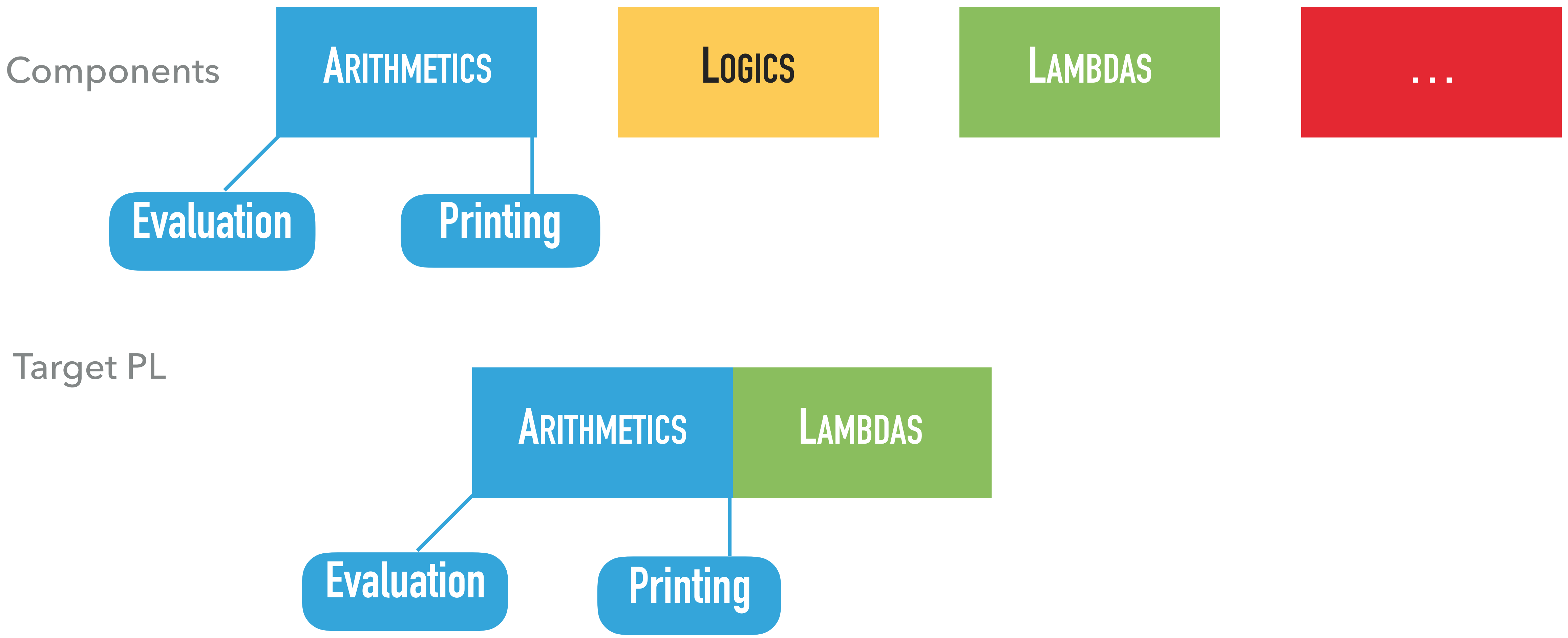
Language Components

Components

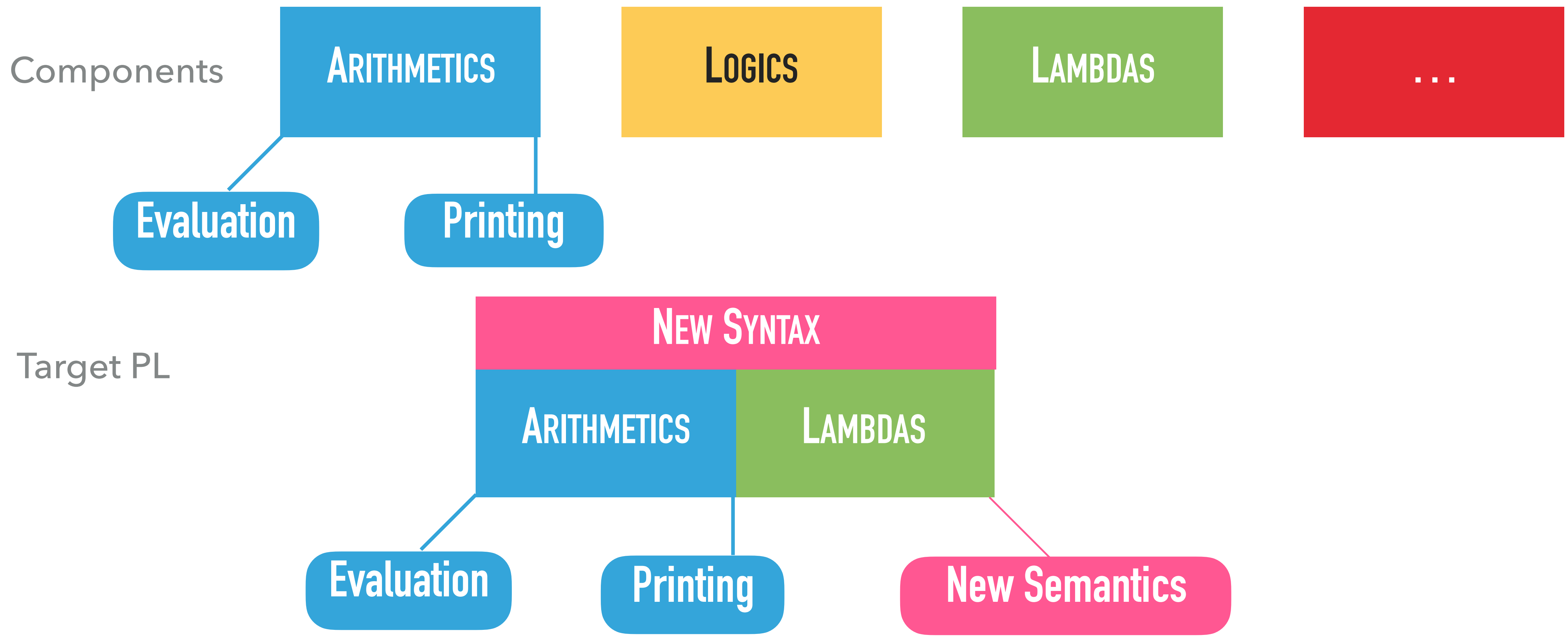
Target PL



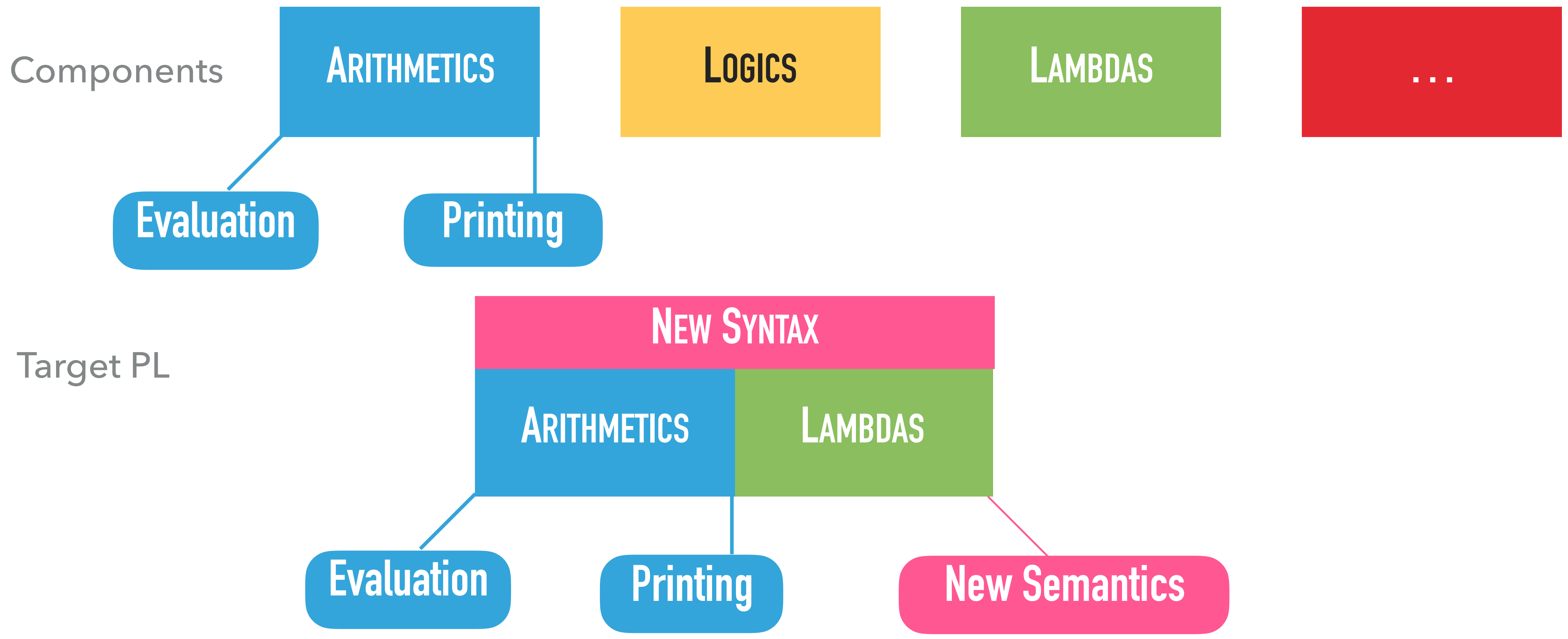
Language Components



Language Components



Language Components



- Developing PLs via composing **language components** with high **reusability** and **extensibility**
  - high reusability reduces the initial effort
  - high extensibility reduces the effort of change

Towards Modularity

Towards Modularity

- Copy & paste
  - code duplication
  - synchronization problem

Towards Modularity

- Copy & paste
  - code duplication
  - synchronization problem
- Syntactic modularity
  - separate file
  - textual composition

Towards Modularity

- Copy & paste
  - code duplication
  - synchronization problem
- Syntactic modularity
  - separate file
  - textual composition
- Semantic modularity
  - separate compilation
  - modular type checking
  - …

Towards Modularity

- Copy & paste
  - code duplication
  - synchronization problem
- Syntactic modularity
  - separate file
  - textual composition
- Semantic modularity
  - separate compilation
  - modular type checking
  - ...

The "expression problem" [Wadler, '98]

Object Algebras

- Object Algebras [Oliveira & Cook, ECOOP'12] are a solution to the expression problem that work in Java-like languages
- However, Object Algebras force a programming style similar to Church encodings/folds
  - lack of control in traversal
  - hard to deal with dependencies
- There are workarounds [Oliveira et al., ECOOP'13; Rendel et al., OOPSLA'14], but
  - complex
  - penalized in performance
  - requiring fancy type system features not available in Java

Contributions

- A new approach to **modular** external visitors
- Simpler modular **dependent** operations
- **Generalized** generic queries and transformations
- Code generation for AST boilerplate code
- Implementation and "Types and Programming Languages" case study

Object Algebras, Internal Visitors and External Visitors

Object Algebras, Internal Visitors and External Visitors

$$e ::= i \mid e + e$$

Object Algebras, Internal Visitors and External Visitors

Internal Visitors / Object Algebras

e ::= i | e + e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
interface Exp {
  <E> E accept(Alg<E> v);
}
class Lit implements Exp {
  int n;
  public <E> E accept(Alg<E> v) {
    return v.Lit(n);
  }
}
class Add implements Exp {
  Exp e1, e2;
  public <E> E accept(Alg<E> v) {
    return v.Add(e1.accept(v), e2.accept(v));
  }
}
class Eval implements Alg<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

Visitor

Element

ConcreteElement

ConcreteVisitor

Object Algebras, Internal Visitors and External Visitors

e ::= i | e + e

Visitor

Element

ConcreteElement

ConcreteVisitor

Internal Visitors / Object Algebras

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
interface Exp {
  <E> E accept(Alg<E> v);
}
class Lit implements Exp {
  int n;
  public <E> E accept(Alg<E> v) {
    return v.Lit(n);
  }
}
class Add implements Exp {
  Exp e1, e2;
  public <E> E accept(Alg<E> v) {
    return v.Add(e1.accept(v), e2.accept(v));
  }
}
class Eval implements Alg<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

External Visitors

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}
interface EExp {
  <E> E accept(EVis<E> v);
}
class ELit implements EExp {
  int n;
  public <E> E accept(EVis<E> v) {
    return v.Lit(n);
  }
}
class EAdd implements EExp {
  EExp e1, e2;
  public <E> E accept(EVis<E> v) {
    return v.Add(e1, e2);
  }
}

class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

Internal Visitors

$$e ::= i \mid e + e$$

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    return i;
  }

  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

Internal Visitors

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    return i;
  }

  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

Internal Visitors

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    return i;
  }

  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

Internal Visitors

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    return i;
  }

  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

```
class ExtEval extends Eval implements ExtAlg<Integer> {
  public Integer Sub(Integer e1, Integer e2) {
    return e1 - e2;
  }
  public Integer If(Integer e1, Integer e2, Integer e3) {
    return !e1.equals(0) ? e2 : e3;
  }
}
```

Internal Visitors

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    return i;
  }

  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

MODULAR but WRONG

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

```
class ExtEval extends Eval implements ExtAlg<Integer> {
  public Integer Sub(Integer e1, Integer e2) {
    return e1 - e2;
  }
  public Integer If(Integer e1, Integer e2, Integer e3) {
    return !e1.equals(0) ? e2 : e3;
  }
}
```

Internal Visitors

e ::= i | e + e | e - e | if e then e else e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    return i;
  }

  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

MODULAR but WRONG

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

```
class ExtEval extends Eval implements ExtAlg<Integer> {
  public Integer Sub(Integer e1, Integer e2) {
    return e1 - e2;
  }
  public Integer If(Integer e1, Integer e2, Integer e3) {
    return !e1.equals(0) ? e2 : e3;
  }
}
```

Internal Visitors

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface Alg<E> {
  E Lit(int i);
  E Add(E e1, E e2);
}
```

```
class Eval implements Alg<Integer> {
  public Integer Lit(int i) {
    System.out.println(i);
    return i;
  }
  public Integer Add(Integer e1, Integer e2) {
    return e1 + e2;
  }
}
```

MODULAR but WRONG

```
interface ExtAlg<E> extends Alg<E> {
  E Sub(E e1, E e2);
  E If(E e1, E e2, E e3);
}
```

```
class ExtEval extends Eval implements ExtAlg<Integer> {
  public Integer Sub(Integer e1, Integer e2) {
    return e1 - e2;
  }
  public Integer If(Integer e1, Integer e2, Integer e3) {
    return !e1.equals(0) ? e2 : e3;
  }
}
```

External Visitors

$$e ::= i \mid e + e$$

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}

interface EExp {
  <E> E accept(EVis<E> v);
}
```

```
class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

External Visitors

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}

interface EExp {
  <E> E accept(EVis<E> v);
}
```

```
class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

External Visitors

$$e ::= i \mid e + e \mid e - e \mid \textbf{if } e \textbf{ then } e \textbf{ else } e$$

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}

interface EExp {
  <E> E accept(EVis<E> v);
}
```

```
class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

```
interface MVis<E> {
  E Lit(int i);
  E Add(MExp e1, MExp e2);
  E Sub(MExp e1, MExp e2);
  E If(MExp e1, MExp e2, MExp e3);
}

interface MExp {
  <E> E accept(MVis<E> v);
}
```

External Visitors

e ::= i | e + e | e - e | if e then e else e

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}

interface EExp {
  <E> E accept(EVis<E> v);
}
```

```
class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

```
interface MVis<E> {
  E Lit(int i);
  E Add(MExp e1, MExp e2);
  E Sub(MExp e1, MExp e2);
  E If(MExp e1, MExp e2, MExp e3);
}

interface MExp {
  <E> E accept(MVis<E> v);
}
```

```
class MEval implements MVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(MExp e1, MExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
  public Integer Sub(MExp e1, MExp e2) {
    return e1.accept(this) - e2.accept(this);
  }
  public Integer If(MExp e1, MExp e2, MExp e3) {
    return !e1.accept(this).equals(0) ?
      e2.accept(this) : e3.accept(this);
  }
}
```

External Visitors

e ::= i | e + e | e - e | if e then e else e

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}

interface EExp {
  <E> E accept(EVis<E> v);
}
```

```
class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

CORRECT but NON-MODULAR

```
interface MVis<E> {
  E Lit(int i);
  E Add(MExp e1, MExp e2);
  E Sub(MExp e1, MExp e2);
  E If(MExp e1, MExp e2, MExp e3);
}

interface MExp {
  <E> E accept(MVis<E> v);
}
```

```
class MEval implements MVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(MExp e1, MExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
  public Integer Sub(MExp e1, MExp e2) {
    return e1.accept(this) - e2.accept(this);
  }
  public Integer If(MExp e1, MExp e2, MExp e3) {
    return !e1.accept(this).equals(0) ?
      e2.accept(this) : e3.accept(this);
  }
}
```

External Visitors

e ::= i | e + e | e - e | if e then e else e

```
interface EVis<E> {
  E Lit(int i);
  E Add(EExp e1, EExp e2);
}

interface EExp {
  <E> E accept(EVis<E> v);
}
```

```
class EEval implements EVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(EExp e1, EExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
}
```

CORRECT but NON-MODULAR

```
interface MVis<E> {
  E Lit(int i);
  E Add(MExp e1, MExp e2);
  E Sub(MExp e1, MExp e2);
  E If(MExp e1, MExp e2, MExp e3);
}

interface MExp {
  <E> E accept(MVis<E> v);
}
```

```
class MEval implements MVis<Integer> {
  public Integer Lit(int i) { return i; }
  public Integer Add(MExp e1, MExp e2) {
    return e1.accept(this) + e2.accept(this);
  }
  public Integer Sub(MExp e1, MExp e2) {
    return e1.accept(this) - e2.accept(this);
  }
  public Integer If(MExp e1, MExp e2, MExp e3) {
    return !e1.accept(this).equals(0) ?
      e2.accept(this) : e3.accept(this);
  }
}
```

Modular External Visitors: Key Idea

$$e ::= i \mid e + e$$

Modular External Visitors: Key Idea

$$e ::= i \mid e + e$$

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

Modular External Visitors: Key Idea

e ::= i | e + e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

Modular External Visitors: Key Idea

e ::= i | e + e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

Modular External Visitors: Key Idea

e ::= i | e + e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

Modular External Visitors: Key Idea

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

Modular External Visitors: Key Idea

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

```
interface AVisExt<R,E> extends AVis<R,E> {
  E Sub(R e1, R e2);
  E If(R e1, R e2, R e3);
}
```

Modular External Visitors: Key Idea

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

```
interface AVisExt<R,E> extends AVis<R,E> {
  E Sub(R e1, R e2);
  E If(R e1, R e2, R e3);
}
```

```
interface AEvalExt<R>
    extends AEval<R>, AVisExt<R,Integer> {
  default Integer Sub(R e1, R e2) {
    return visitExp(e1) - visitExp(e2);
  }
  default Integer If(R e1, R e2, R e3) {
    return !visitExp(e1).equals(0) ?
      visitExp(e2) : visitExp(e3);
  }
}
```

Modular External Visitors: Key Idea

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

MODULAR and CORRECT

```
interface AVisExt<R,E> extends AVis<R,E> {
  E Sub(R e1, R e2);
  E If(R e1, R e2, R e3);
}
```

```
interface AEvalExt<R>
    extends AEval<R>, AVisExt<R,Integer> {
  default Integer Sub(R e1, R e2) {
    return visitExp(e1) - visitExp(e2);
  }
  default Integer If(R e1, R e2, R e3) {
    return !visitExp(e1).equals(0) ?
      visitExp(e2) : visitExp(e3);
  }
}
```

Modular External Visitors: Key Idea

e ::= i | e + e | e - e | **if** e **then** e **else** e

```
interface AVis<R,E> {
  E Lit(int i);
  E Add(R e1, R e2);
  E visitExp(R e);
}
```

```
interface AEval<R> extends AVis<R,Integer> {
  default Integer Lit(int i) { return i; }
  default Integer Add(R e1, R e2) {
    return visitExp(e1) + visitExp(e2);
  }
}
```

MODULAR and CORRECT

```
interface AVisExt<R,E> extends AVis<R,E> {
  E Sub(R e1, R e2);
  E If(R e1, R e2, R e3);
}
```

```
interface AEvalExt<R>
    extends AEval<R>, AVisExt<R,Integer> {
  default Integer Sub(R e1, R e2) {
    return visitExp(e1) - visitExp(e2);
  }
  default Integer If(R e1, R e2, R e3) {
    return !visitExp(e1).equals(0) ?
     visitExp(e2) : visitExp(e3);
  }
}
```

Modular External Visitors: Instantiation and Client Code

e ::= i | e + e

Modular External Visitors: Instantiation and Client Code

e ::= i | e + e

```
interface CExp {
  <E> E accept(AVis<CExp,E> v);
}
class CLit implements CExp {…}
class CAdd implements CExp {
  CExp e1, e2;
  public CAdd(CExp e1, CExp e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Add(e1, e2);
  }
}
```

Modular External Visitors: Instantiation and Client Code

e ::= i | e + e

```
interface CExp {
  <E> E accept(AVis<CExp,E> v);
}
class CLit implements CExp {…}
class CAdd implements CExp {
  CExp e1, e2;
  public CAdd(CExp e1, CExp e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Add(e1, e2);
  }
}
interface CVis<E> extends AVis<CExp,E> {
  default E visitExp(CExp e) {
    return e.accept(this);
  }
}
```

Modular External Visitors: Instantiation and Client Code

e ::= i | e + e

```
interface CExp {
  <E> E accept(AVis<CExp,E> v);
}
class CLit implements CExp {…}
class CAdd implements CExp {
  CExp e1, e2;
  public CAdd(CExp e1, CExp e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Add(e1, e2);
  }
}

interface CVis<E> extends AVis<CExp,E> {
  default E visitExp(CExp e) {
    return e.accept(this);
  }
}

class CEval implements AEval<CExp>, CVis<Integer> {}

CExp e = new CAdd(new CLit(1), new CLit(2));
e.accept(new CEval()); // 3
```

Modular External Visitors: Instantiation and Client Code

e ::= i | e + e

```
interface CExp {
  <E> E accept(AVis<CExp,E> v);
}
class CLit implements CExp {…}
class CAdd implements CExp {
  CExp e1, e2;
  public CAdd(CExp e1, CExp e2) {
    this.e1 = e1; this.e2 = e2;
  }
  public <E> E accept(AVis<CExp,E> v) {
    return v.Add(e1, e2);
  }
}

interface CVis<E> extends AVis<CExp,E> {
  default E visitExp(CExp e) {
    return e.accept(this);
  }
}
```

e ::= i | e + e | e - e | if e then e else e

```
interface CExpExt {
  <E> E accept(AVisExt<CExpExt,E> v);
}

interface CVisExt<E> extends AVisExt<CExpExt,E> {
  default E visitExp(CExpExt e) {
    return e.accept(this);
  }
}

//... 4 AST classes elided including Lit and Add
```

```
class CEval implements AEval<CExp>, CVis<Integer> {}

CExp e = new CAdd(new CLit(1), new CLit(2));
e.accept(new CEval()); // 3
```

A Summary On Object Algebras, Internal Visitors and External Visitors

Approach	**Modular Visitor**	**Modular AST**	**Traversal Control**
Object Algebras	Yes	Yes	No
Internal Visitors	Yes	No	No
External Visitors	No	No	Yes
Modular External Visitors	**Yes**	**No**	**Yes**

A Summary On Object Algebras, Internal Visitors and External Visitors

Approach	Modular Visitor	Modular AST	Traversal Control
Object Algebras	Yes	Yes	No
Internal Visitors	Yes	No	No
External Visitors	No	No	Yes
Modular External Visitors	**Yes**	**No**	**Yes**

Mechanical

EVF for Modularity and Reuse of PL Implementations

- **EVF** is an annotation processor that generates boilerplate code related to modular external visitors
  - AST infrastructure
  - traversal templates generalizing on **Shy** [Zhang et al., OOPSLA'15]
- Usage
  - annotating Object Algebra interfaces with `@Visitor` and that's it!

Untyped Lambda Calculus: Syntax

e	$::=$	x	variable
		$\lambda x.e$	abstraction
		$e\ e$	application
		i	literal
		$e - e$	subtraction

Untyped Lambda Calculus: Syntax

e	$::=$	x	variable
		$\lambda x.e$	abstraction
		$e\ e$	application
		i	literal
		$e - e$	subtraction

```
@Visitor
interface LamAlg<Exp> {
  Exp Var(String x);
  Exp Abs(String x, Exp e);
  Exp App(Exp e1, Exp e2);
  Exp Lit(int i);
  Exp Sub(Exp e1, Exp e2);
}
```

Untyped Lambda Calculus: Syntax

e	$::=$	x	variable
		$\lambda x.e$	abstraction
		$e\ e$	application
		i	literal
		$e - e$	subtraction

```
interface GLamAlg<Exp, OExp> {
  OExp App(Exp p1, Exp p2);
  OExp Sub(Exp p1, Exp p2);
  OExp Abs(String p1, Exp p2);
  OExp Var(String p1);
  OExp Lit(int p1);
  OExp visitExp(Exp e);
}
```

Untyped Lambda Calculus: Free Variables

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.e) & = & FV(e) \setminus \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(i) & = & \varnothing \\
FV(e_1 - e_2) & = & FV(e_1) \cup FV(e_2)
\end{array}
$$

Untyped Lambda Calculus: Free Variables

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.e) & = & FV(e) \setminus \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(i) & = & \varnothing \\
FV(e_1 - e_2) & = & FV(e_1) \cup FV(e_2)
\end{array}
$$

Query :: Exp → Set<String>

Untyped Lambda Calculus: Free Variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \varnothing \\
FV(e_1 - e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Query :: Exp → Set<String>

Untyped Lambda Calculus: Free Variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \varnothing \\
FV(e_1 - e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Query :: Exp → Set<String>

```
interface FreeVars<Exp> extends LamAlgQuery<Exp, Set<String>> {
  default Monoid<Set<String>> m() {
    return new SetMonoid<>();
  }
  default Set<String> Var(String x) {
    return Collections.singleton(x);
  }
  default Set<String> Abs(String x, Exp e) {
    return visitExp(e).stream().filter(y -> !y.equals(x))
      .collect(Collectors.toSet());
  }
}
```

Untyped Lambda Calculus: Free Variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \varnothing \\
FV(e_1 - e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Query :: Exp → Set<String>

```
interface FreeVars<Exp> extends LamAlgQuery<Exp, Set<String>> {
}
```

```
interface LamAlgQuery<Exp, O> extends GLamAlg<Exp, O> {
  Monoid<O> m();

  default O Var(String x) { return m().empty(); }
  default O Abs(String x, Exp e) { return visitExp(e); }
  default O App(Exp e1, Exp e2) {
    return Stream.of(visitExp(e1), visitExp(e2)).reduce(m().empty(), m()::join);
  }
  default O Lit(int i) { return m().empty(); }
  default O Sub(Exp e1, Exp e2) {
    return Stream.of(visitExp(e1), visitExp(e2)).reduce(m().empty(), m()::join);
  }
}
```

Untyped Lambda Calculus: Free Variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \varnothing \\
FV(e_1 - e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Query :: Exp → Set<String>

```
interface FreeVars<Exp> extends LamAlgQuery<Exp, Set<String>> {
  default Monoid<Set<String>> m() {
    return new SetMonoid<>();
  }
  default Set<String> Var(String x) {
    return Collections.singleton(x);
  }
  default Set<String> Abs(String x, Exp e) {
    return visitExp(e).stream().filter(y -> !y.equals(x))
      .collect(Collectors.toSet());
  }
}
```

Untyped Lambda Calculus: Free Variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \varnothing \\
FV(e_1 - e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Query :: Exp → Set<String>

```
interface FreeVars<Exp> extends LamAlgQuery<Exp, Set<String>> {
  default Monoid<Set<String>> m() {
    return new SetMonoid<>();
  }
```

```
class SetMonoid<T> implements Monoid<Set<T>> {
  public Set<T> empty() { return Collections.emptySet(); }
  public Set<T> join(Set<T> x, Set<T> y) {
    return Stream.concat(x.stream(), y.stream()).collect(Collectors.toSet());
  }
}
```

```
  }
}
```

Untyped Lambda Calculus: Free Variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(i) &= \varnothing \\
FV(e_1 - e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

Query :: Exp → Set<String>

```java
interface FreeVars<Exp> extends LamAlgQuery<Exp, Set<String>> {
  default Monoid<Set<String>> m() {
    return new SetMonoid<>();
  }
  default Set<String> Var(String x) {
    return Collections.singleton(x);
  }
  default Set<String> Abs(String x, Exp e) {
    return visitExp(e).stream().filter(y -> !y.equals(x))
      .collect(Collectors.toSet());
  }
}
```

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1\ - [x \mapsto s]e_2 &
\end{array}
$$

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1\ - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1\ - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1\ - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
  String x();
  Exp s();
  Set<String> FV(Exp e);

  default Exp Var(String y) {
    return y.equals(x()) ? s() : alg().Var(y);
  }
  default Exp Abs(String y, Exp e) {
    if (y.equals(x())) return alg().Abs(y, e);
    if (FV(s()).contains(y)) throw new RuntimeException();
    return alg().Abs(y, visitExp(e));
  }
}
```

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \land y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1 - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
  String x();
  Exp s();
  Set<String> FV(Exp e);

  default Exp Var(String y) {
    return y.equals(x()) ? s() : alg().Var(y);
  }
  default Exp Abs(String y, Exp e) {
    if (y.equals(x())) return alg().Abs(y, e);
    if (FV(s()).contains(y)) throw new RuntimeException();
    return alg().Abs(y, visitExp(e));
  }
}
```

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1 - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
```

```
interface LamAlgTransform<Exp> extends GLamAlg<Exp, Exp> {
  LamAlg<Exp> alg();
  default Exp Var(String x) { return alg().Var(x); }
  default Exp Abs(String x, Exp e) { return alg().Abs(x, visitExp(e)); }
  default Exp App(Exp e1, Exp e2) { return alg().App(visitExp(e2), visitExp(e2)); }
  default Exp Lit(int i) { return alg().Lit(i); }
  default Exp Sub(Exp e1, Exp e2) { return alg().Sub(visitExp(e1), visitExp(e2)); }
}
```

```
      if (y.equals(x())) return alg().Abs(y, e);
      if (FV(s()).contains(y)) throw new RuntimeException();
      return alg().Abs(y, visitExp(e));
    }
}
```

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1 - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
  String x();
  Exp s();
  Set<String> FV(Exp e);

  default Exp Var(String y) {
    return y.equals(x()) ? s() : alg().Var(y);
  }
  default Exp Abs(String y, Exp e) {
    if (y.equals(x())) return alg().Abs(y, e);
    if (FV(s()).contains(y)) throw new RuntimeException();
    return alg().Abs(y, visitExp(e));
  }
}
```

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y && \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) &= \lambda x.e \\
[x \mapsto s](\lambda y.e) &= \lambda y.[x \mapsto s]e && \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) &= [x \mapsto s]e_1\ [x \mapsto s]e_2 \\
[x \mapsto s]i &= i \\
[x \mapsto s](e_1 - e_2) &= [x \mapsto s]e_1 - [x \mapsto s]e_2
\end{aligned}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
  String x();
  Exp s();
  Set<String> FV(Exp e);

  default Exp Var(String y) {
    return y.equals(x()) ? s() : alg().Var(y);
  }
  default Exp Abs(String y, Exp e) {
    if (y.equals(x())) return alg().Abs(y, e);
    if (FV(s()).contains(y)) throw new RuntimeException();
    return alg().Abs(y, visitExp(e));
  }
}
```

Dependency Declaration

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \wedge y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1 - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
  String x();
  Exp s();
  Set<String> FV(Exp e);

  default Exp Var(String y) {
    return y.equals(x()) ? s() : alg().Var(y);
  }
  default Exp Abs(String y, Exp e) {
    if (y.equals(x())) return alg().Abs(y, e);
    if (FV(s()).contains(y)) throw new RuntimeException();
    return alg().Abs(y, visitExp(e));
  }
}
```

Dependency Declaration

Dependency Usage

Untyped Lambda Calculus: Capture-avoiding Substitution

$$
\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda x.e) & = & \lambda x.e & \\
[x \mapsto s](\lambda y.e) & = & \lambda y.[x \mapsto s]e & \text{if } y \neq x \land y \notin FV(s) \\
[x \mapsto s](e_1\ e_2) & = & [x \mapsto s]e_1\ [x \mapsto s]e_2 & \\
[x \mapsto s]i & = & i & \\
[x \mapsto s](e_1 - e_2) & = & [x \mapsto s]e_1 - [x \mapsto s]e_2 &
\end{array}
$$

Transformation :: (Exp, String, Exp) → Exp

```
interface SubstVar<Exp> extends LamAlgTransform<Exp> {
  String x();
  Exp s();
  Set<String> FV(Exp e);

  default Exp Var(String y) {
    return y.equals(x()) ? s() : alg().Var(y);
  }
  default Exp Abs(String y, Exp e) {
    if (y.equals(x())) return alg().Abs(y, e);
    if (FV(s()).contains(y)) throw new RuntimeException();
    return alg().Abs(y, visitExp(e));
  }
}
```

Dependency Declaration

Dependency Usage

Traversal Control

Untyped Lambda Calculus: Instantiation and Client Code

Instantiation

```
class FreeVarsImpl implements FreeVars<CExp>, LamAlgVisitor<Set<String>> {}
class SubstVarImpl implements SubstVar<CExp>, LamAlgVisitor<CExp> {
  String x;
  CExp s;
  public SubstVarImpl(String x, CExp s) { this.x = x; this.s = s; }
  public String x() { return x; }
  public CExp s() { return s; }
  public Set<String> FV(CExp e) { return new FreeVarsImpl().visitExp(e); }
  public LamAlg<CExp> alg() { return new LamAlgFactory(); }
}
```

Untyped Lambda Calculus: Instantiation and Client Code

Instantiation

```
class FreeVarsImpl implements FreeVars<CExp>, LamAlgVisitor<Set<String>> {}
class SubstVarImpl implements SubstVar<CExp>, LamAlgVisitor<CExp> {
  String x;
  CExp s;
  public SubstVarImpl(String x, CExp s) { this.x = x; this.s = s; }
  public String x() { return x; }
  public CExp s() { return s; }
  public Set<String> FV(CExp e) { return new FreeVarsImpl().visitExp(e); }
  public LamAlg<CExp> alg() { return new LamAlgFactory(); }
}
```

Client code

```
LamAlgFactory alg = new LamAlgFactory();
CExp exp = alg.App(alg.Abs("y", alg.Var("y")), alg.Var("x")); // (\y.y) x
new FreeVarsImpl().visitExp(exp); // {"x"}
new SubstVarImpl("x", alg.Lit(1)).visitExp(exp); // (\y.y) 1
```

A Comparison with Other Implementations

Approach	**Modular**	**Syntax**	**Free Variables**		**Substitution**	
		SLOC	**SLOC**	**# Cases**	**SLOC**	**# Cases**
The VISITOR Pattern	No	46	20	5	22	5
Object Algebras (w/ **Shy**)	Yes	7	12	2	55	5
EVF	Yes	7	12	2	13	2

Reusing the Untyped Lambda Calculus as an Language Component

Reusing the Untyped Lambda Calculus as an Language Component

```
@Visitor
interface ExtLamAlg<Exp> extends LamAlg<Exp> {
  Exp Bool(boolean b);
  Exp If(Exp e1, Exp e2, Exp e3);
}
```

Reusing the Untyped Lambda Calculus as an Language Component

```
@Visitor
interface ExtLamAlg<Exp> extends LamAlg<Exp> {
  Exp Bool(boolean b);
  Exp If(Exp e1, Exp e2, Exp e3);
}
interface ExtFreeVars<Exp> extends ExtLamAlgQuery<Exp,Set<String>>, FreeVars<Exp> {}

interface ExtSubstVar<Exp> extends ExtLamAlgTransform<Exp>, SubstVar<Exp> {}
```

Reusing the Untyped Lambda Calculus as an Language Component

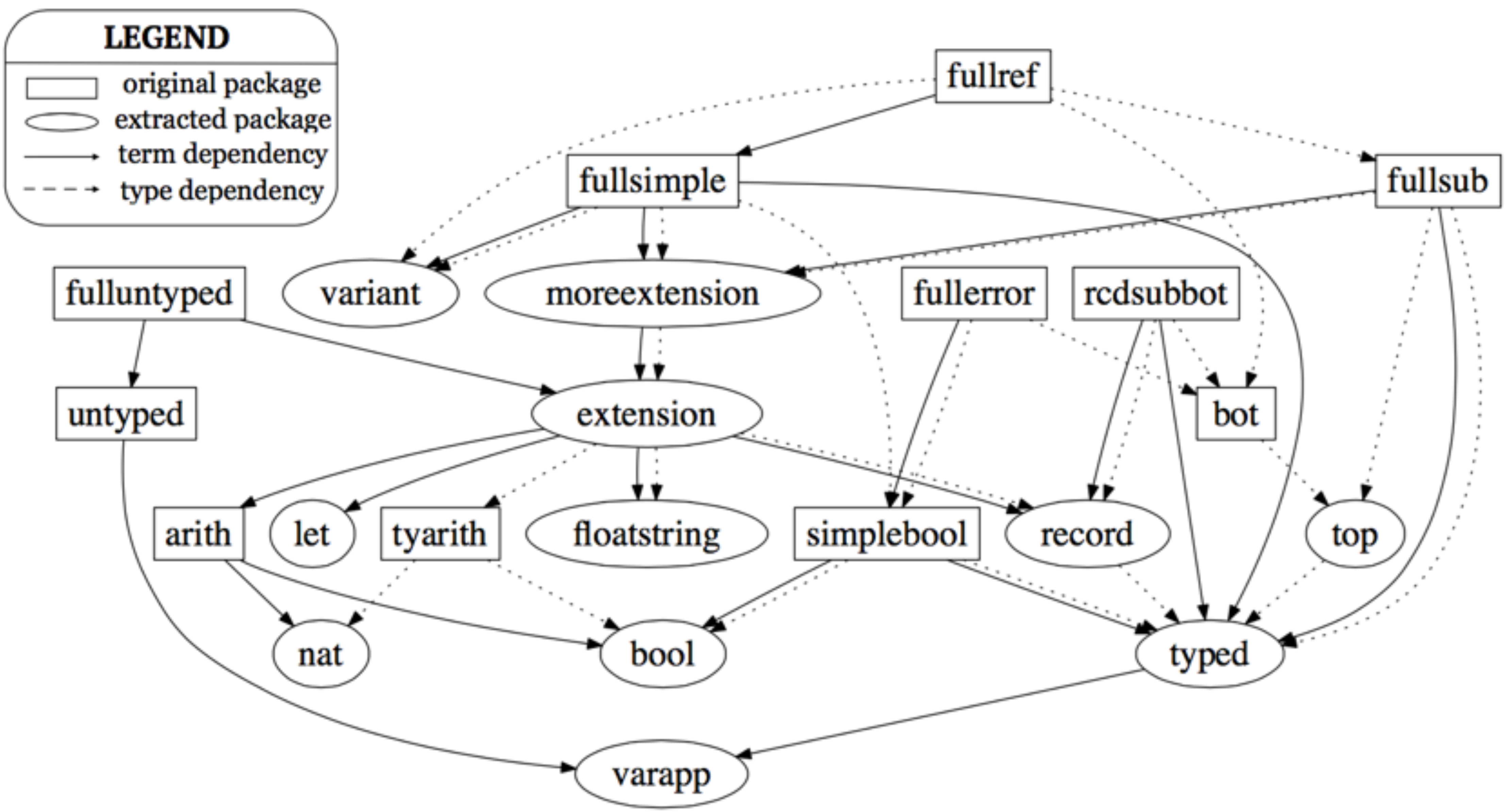
```
@Visitor
interface ExtLamAlg<Exp> extends LamAlg<Exp> {
  Exp Bool(boolean b);
  Exp If(Exp e1, Exp e2, Exp e3);
}
interface ExtFreeVars<Exp> extends ExtLamAlgQuery<Exp,Set<String>>, FreeVars<Exp> {}

interface ExtSubstVar<Exp> extends ExtLamAlgTransform<Exp>, SubstVar<Exp> {}
```

- Reduction of implementation effort
  - reuse from extensibility
  - reuse from traversal templates
- Reduction of knowledge about PL implementations
  - technical details are encapsulated

Case Study: Overview

- Refactoring a large number of non-modular interpreters from the "Types and Programming Languages" book



Case Study: Evaluation

Extracted Package	EVF	Original Package	EVF	OCaml	% Reduced
bool	98	arith	33	102	68%
extension	34	bot	61	184	67%
floatstring	104	fullerror	105	366	72%
let	47	fullref	247	880	72%
moreextension	106	fullsimple	83	651	88%
nat	103	fullsub	116	628	82%
record	198	fulluntyped	47	300	85%
top	86	rcdsubbot	39	255	85%
typed	138	simplebool	38	211	77%
utils	172	tyarith	26	135	78%
varapp	65	untyped	46	128	61%
variant	161	**Total**	**2153**	**3840**	**44%**

Component	EVF	OCaml	% Reduced
AST Definition	85	231	64%
Small-step Evaluator	263	481	46%

Related Work

- Extensible visitors
- Structure-shy traversals with visitors
- Object Algebras and Church encodings
- Component-based language development
- Language workbenches
- Software product-lines

Summary

- We have presented an modular external visitor encoding
  - workable for Java-like languages
  - allowing dependencies to be expressed modularly
  - providing users with flexible traversal strategies
- We have presented the **EVF** framework
  - generates boilerplate code including ASTs and AST traversals
- Evaluated artifacts are available at
  - https://github.wxzh/EVF

Summary

- We have presented an modular external visitor encoding
  - workable for Java-like languages
  - allowing dependencies to be expressed modularly
  - providing users with flexible traversal strategies
- We have presented the **EVF** framework
  - generates boilerplate code including ASTs and AST traversals
- Evaluated artifacts are available at
  - https://github.wxzh/EVF

Thank you!

Performance Measurements

- A microbenchmark summing up a list of length 2000 for 10,000 times [Palsberg & Jay, COMPSAC'98]

Approach	**Time (ms)**
Imperative Visitor	133
Functional Visitor	163
Runabout	278
EVF	262

- The performance penalty comes from one more level indirection introduced by `visitExp`