香 港 大 學
THE UNIVERSITY OF HONG KONG

Semantic Modularization Techniques in Practice: A TAPL case study

Bruno C. d. S. Oliveira

Joint work with Weixin Zhang, Haoyuan Zhang and Huang Li

July 17, 2017

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

Weixin Zhang and Bruno C. d. S. Oliveira

(ECOOP 2017)


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

Weixin Zhang and Bruno C. d. S. Oliveira

Type-safe Modular Parsing

Haoyuan Zhang, Huang Li and Bruno C. d. S. Oliveira

Submitted


Type-Safe Modular Parsing

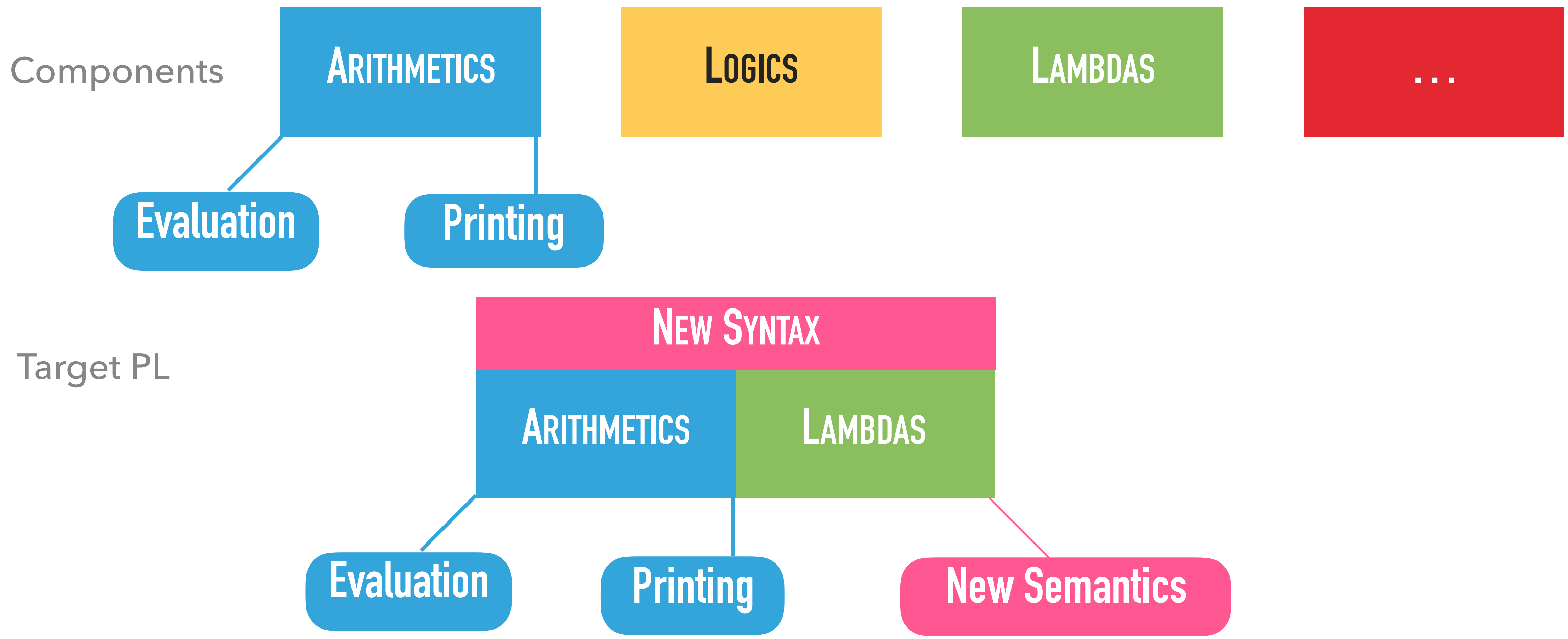
This Talk

- Presents work on **semantic modularity** techniques based on variants of Object Algebras/Modular Visitors;
- Showing that such techniques can **scale** beyond tiny problems (such as Wadler's Expression Problem);
- Case studies that reimplement "**Types and Programming Languages**" (TAPL) interpreters using such semantically modular techniques. Covers: **semantics** and **parsing;**
- **Not in the talk:** I will not cover in detail the coding techniques themselves. Rather I'll focus on the case study results.

Motivation

- New PLs/DSLs are needed; 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 declarations, arithmetic operations …
- But it is **hard to materialize *conceptual reuse* into *software engineering reuse***

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

Modularisation Techniques

Approaches to Modularity: Copy & Paste

- The **most widely used** approach in practice!
- pros: **extremely easy**!
- cons: code **duplication**
- cons: **synchronisation problem/maintenance/ evolution**
    - hard do synchronise changes across copies

Approaches to Modularity: Syntactic Modularity

- Quite popular in Language Workbenches; Software-Product Lines tools
  - Examples: Attribute grammar systems; ASF+SDF; Spoofax; Monticore
- pros: **no code duplication**
- pros: implementable with **relatively simple meta-programming techniques** (textual/ source-code composition); and/or DSLs

Approaches to Modularity: Syntactic Modularity

- cons: lacks some desirable properties:
  - **modular type-checking** (consequently less IDE support)
  - **separate compilation**
  - harder to provide good error messages

Approaches to Modularity: Semantic Modularity

- Typically used as **design patterns** in languages with reasonably expressive type systems
  - Cake Pattern (Scala); Data Types a la Carte (Haskell); Object Algebras (Java/Scala) or Finally Tagless (Haskell/OCaml)
- pros: **naturally supported in the programming language itself**. Therefore we get (for free):
  - **Modular type-checking**
  - **Separate compilation**
  - Other goodies derived from those: better **IDE support/ code-completion**; reasonable **error messages**

Approaches to Modularity: Semantic Modularity

- cons: the coding patterns can be heavy (too many type annotations; **boilerplate code;** PL support is not ideal)
- cons: **not well-proven in practice** (address small challenge problems such as the Expression Problem (Wadler 98))
- **stereotype**: can only solve small problems; too hard to use in practice.

Frameworks for Semantic Modularity

Frameworks for Semantic Modularity: Lets fight the stereotype!

- Our frameworks combine:
  - **lightweight design patterns** for modularity
  - **program generation** techniques to **remove boilerplate** code from such design patterns
  - **libraries of language components** (including parsing, and semantics)
- We have a few Frameworks: **EVF** (for Java), **Parsing Framework** (for Scala), **United framework** (in progress, Scala)

Example: The EVF Java Framework

- **EVF** is an **annotation processor** that generates boilerplate code related to modular external visitors
  - AST infrastructure
  - traversal templates generalising **Shy** [Zhang et al., OOPSLA'15] (Think Adaptive Programming, Stratego or Scrap your Boilerplate)
- Usage
  - annotating Object Algebra interfaces (AST interface) with `@Visitor`
  - Java 8 interfaces with defaults for **multiple inheritance**

Untyped Lambda Calculus: Syntax

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

Annotation-based AST

```
@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: Free Variables

interesting cases

$$FV(x) = \{x\}$$
$$FV(\lambda x.e) = FV(e) \setminus \{x\}$$

boring cases

$$FV(e_1\, e_2) = FV(e_1) \cup FV(e_2)$$
$$FV(i) = \varnothing$$
$$FV(e_1 - e_2) = FV(e_1) \cup FV(e_2)$$

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());
  }
}
```

Structure-Shy Programming

(Past work: Adaptive Programming, Stratego, SyB)

Untyped Lambda Calculus: Capture-avoiding Substitution

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

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

Untyped Lambda Calculus: Capture-avoiding Substitution

```
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: 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

- Results of EVF are better than previous frameworks based on Object Algebras because:
  - EVF **traversals are more flexible** (easy to deal with non-bottom up traversals);
  - EVF has **better support for dependencies**;

Modularity/Extensibility: Reusing the Untyped Lambda Calculus

```
@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

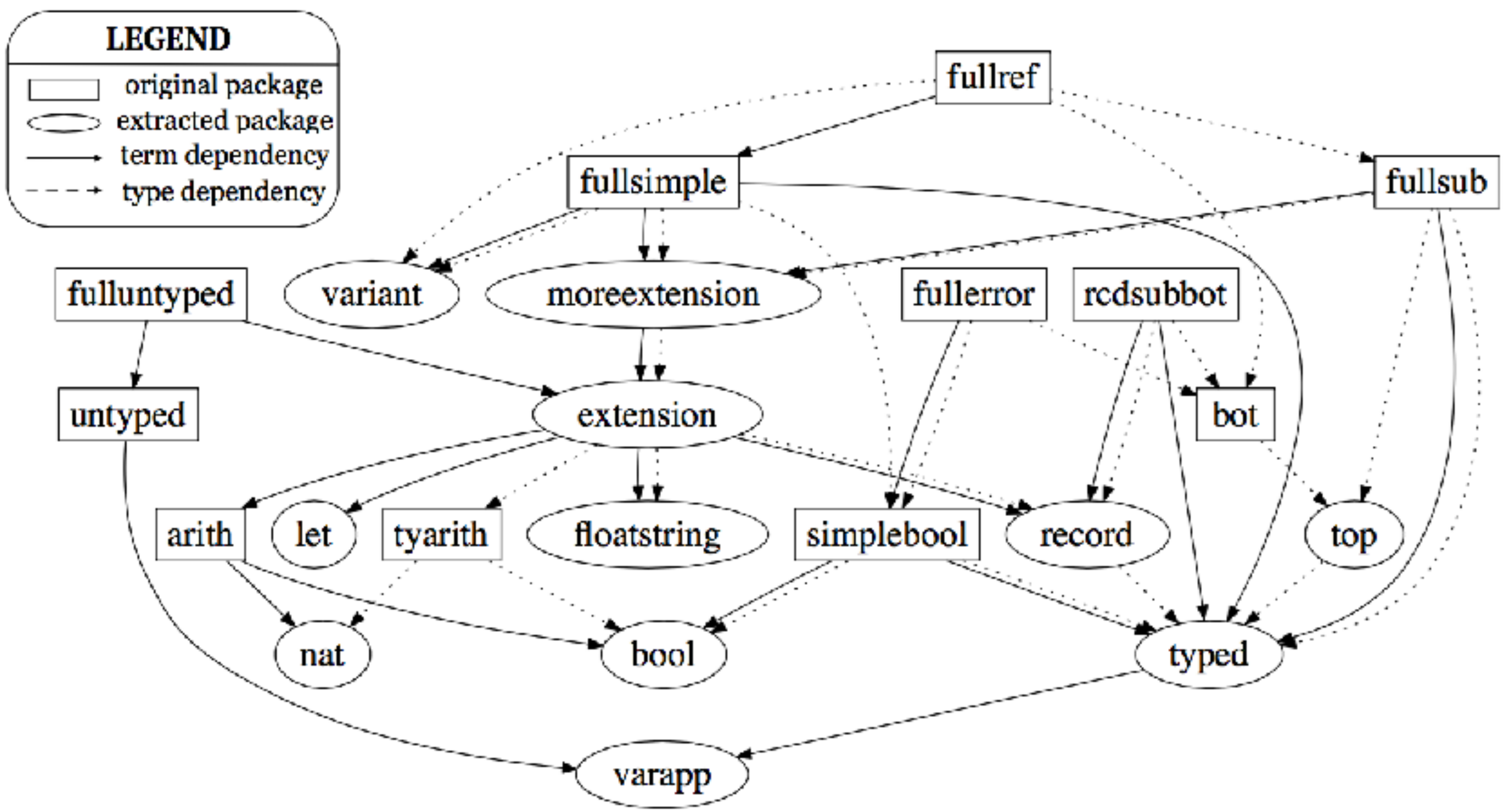
TAPL Case Studies

Why TAPL?

- **Widely used and accepted book** with a large collection of language variants/features
- Several **language features used in practice**
- Implementations (in OCaml) account for different aspects: **dynamic semantics**, **static semantics**, and **parsing**
- **Non-trivial to modularize**:
  - **small-step semantics**
  - **non-compositional operations**
  - many **dependencies**

EVF Case Study: Overview (only semantics)

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



EVF 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%**

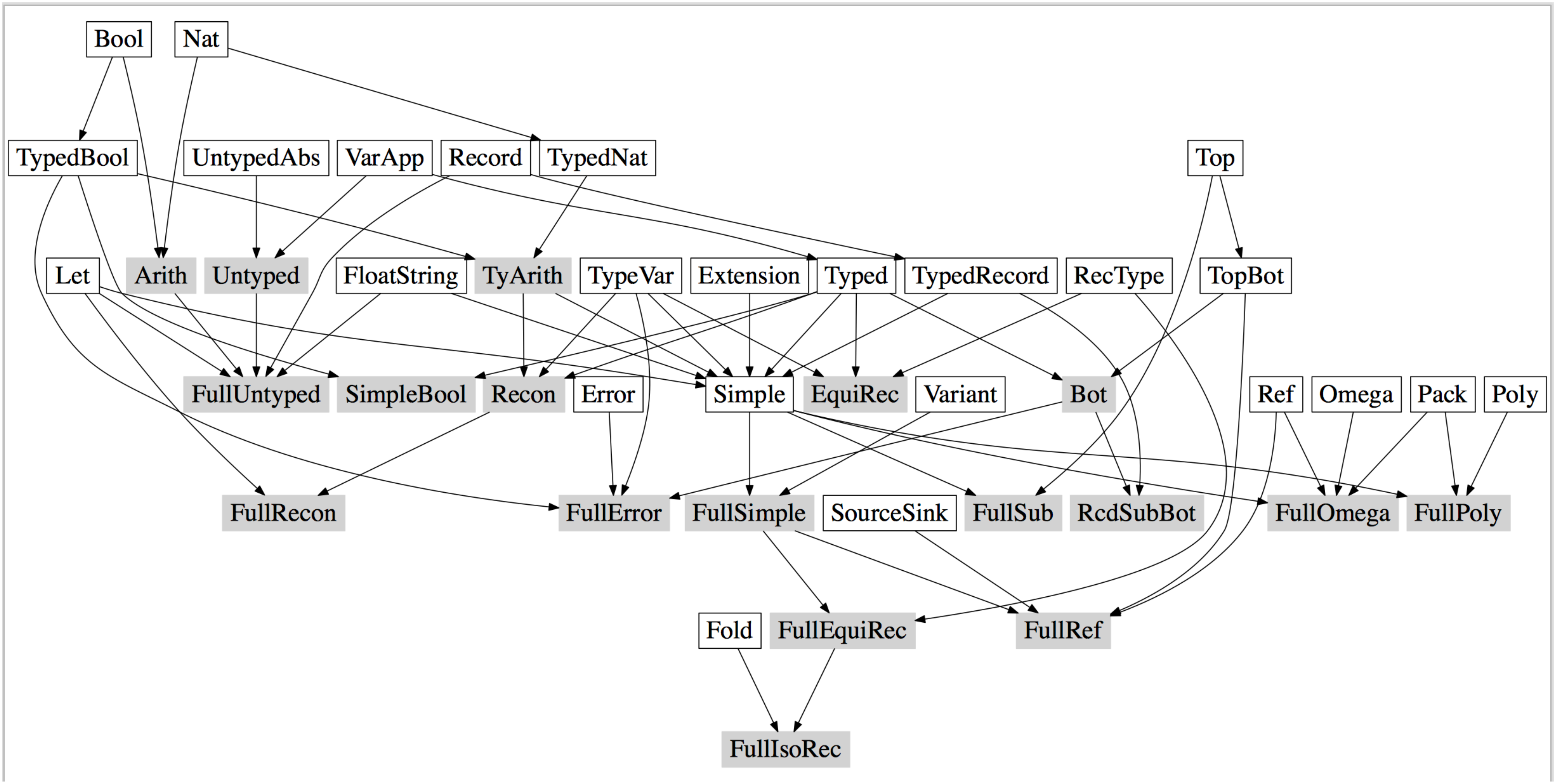
Component	EVF	OCaml	% Reduced
AST Definition	85	231	64%
Small-step Evaluator	263	481	46%

Difficulties

- Modularity
  - **no good support for modular pattern matching** (bad for small step semantics and some operations)
  - **Dependencies** are hard, but **manageable** in EVF
- Drawbacks
  - **Instantiation code is boilerplate**, but still has to be defined manually. Dependencies introduce quite a bit of instantiation boilerplate.
  - Some **coding patterns are still heavy**.

Parsing Case Study: Overview (only syntax)

- Refactoring a 18 parsers for non-modular interpreters from the "Types and Programming Languages" book



Parsing Framework (in Scala)

- Parsing framework combines:
  - **design patterns** for parsing (using Packrat parser combinators and Object Algebras)
  - **libraries of parsing components**
  - **Multiple inheritance** (traits in Scala)
- Supports:
  - **modular type-checking**
  - **separate compilation**
  - **modular (and type-safe) composition** of parsers
- Doesn't support:
  - **ambiguity checking** (as any parser combinator based approach)

Composition: A Simple Example

An example of building the Bot calculus by composition

- Component Typed for simply typed lambda calculus
- Component TopBot for top and bottom types

```
object Bot {
  trait Alg[E, T] extends Typed.Alg[E, T] with TopBot.Alg[T]

  trait Print extends Alg[String, String] with Typed.Print with TopBot.Print

  trait Parse[E, T] extends Typed.Parse[E, T] with TopBot.Parse[T] {
    override val alg: Alg[E, T]
    val pBotE: Parser[E] = pTypedE
    val pBotT: Parser[T] = pTypedT ||| pTopBotT
    override val pE: Parser[E] = pBotE
    override val pT: Parser[T] = pBotT
  }
}
```

Longest match composition

Comparison

Calculus Name	SLOC			Time (ms)		
	NonMod	Mod_{OA}	(+/-)%	NonMod	Mod_{OA}	(+/-)%
Arith	77	77	+0.0	741	913	+23.2
Untyped	48	53	+10.4	770	1018	+32.2
FullUntyped	131	75	-42.7	1297	1854	+42.9
TyArith	89	54	-39.3	746	888	+19.0
SimpleBool	90	42	-53.3	1376	1782	+29.5
FullSimple	244	127	-48.0	1441	2270	+57.5
Bot	87	48	-44.8	1080	1287	+19.2
FullRef	277	65	-76.5	1438	2291	+59.3
FullError	112	41	-63.4	1410	1946	+38.0
RcdSubBot	125	22	-82.4	1247	1524	+22.2
FullSub	225	22	-90.2	1320	1979	+49.9
FullEquiRec	250	36	-85.6	1407	2200	+56.4
FullIsoRec	259	40	-84.6	1492	2253	+51.0
EquiRec	81	22	-72.8	994	1254	+26.2
Recon	138	22	-84.1	1044	1482	+42.0
FullRecon	142	22	-84.5	1094	1645	+50.4
FullPoly	248	68	-72.6	1398	2086	+49.2
FullOmega	315	68	-78.4	1451	2352	+62.1
Total	2938	904	-69.2	21746	31024	+42.7

Comparison (Performance Penalties)

- We did further experiments to identify the performance penalties
  - **Object Algebras vs Case classes** (almost no impact on performance)
  - **longest match combinator (7% slower vs alternative combinator)**
- Main reason for slowdown: **extra method calls**/ dispatching due to modularity (more indirection)
- Future work: **Partial evaluation/staging** to remove indirections

Conclusion

- Semantic modularity techniques can scale reasonably well to small/medium size languages, thanks to:
  - **multiple inheritance** and OO native support for **open recursion**
  - **subtyping** and **generics**
  - **type-refinement** (covariant refinement of return types)
  - **annotation-based code generation**
- Using mainstream languages is not perfect, though:
  - Would be better to have **native language support** for Object Algebras/Modular Visitors
  - Support for some form of **modular pattern matching** is highly desirable
  - Mainstream languages still have **instantiation boilerplate**