# Type-Safe Modular Parsing

Haoyuan Zhang
hyzhang@cs.hku.hk
The University of Hong Kong

Huang Li
hli@cs.hku.hk
The University of Hong Kong

Bruno C. d. S. Oliveira
bruno@cs.hku.hk
The University of Hong Kong

## Abstract

Over the years a lot of effort has been put on solving extensibility problems, while retaining important software engineering properties such as modular type-safety and separate compilation. Most previous work focused on operations that traverse and process extensible Abstract Syntax Tree (AST) structures. However, there is almost no work on operations that *build* such extensible ASTs, including *parsing*.

This paper investigates solutions for the problem of *modular parsing*. We focus on *semantic* modularity and not just *syntactic* modularity. That is, the solutions should not only allow complete parsers to be built out of modular parsing components, but also enable the parsing components to be *modularly type-checked* and *separately compiled*. We present a technique based on parser combinators that enables modular parsing. We show that Packrat parsing techniques, provide solutions for such modularity problems, and enable reasonable performance in a modular setting. Extensibility is achieved using *multiple inheritance* and *Object Algebras*. To evaluate the approach we conduct a case study based on the "Types and Programming Languages" interpreters. The case study shows the effectiveness at reusing parsing code from existing interpreters, and the total parsing code is 69% shorter than an existing code base using a non-modular parsing approach.

## 1 Introduction

The quest for improved modularity, variability and extensibility of programs has been going on since the early days of Software Engineering [30]. Modern Programming Languages (PLs) enable a certain degree of modularity, but they have limitations as illustrated by well-known problems such as the Expression Problem [44]. The Expression Problem refers to the difficulty of writing data abstractions that can be easily extended with both new operations and new data variants. Traditionally the kinds of data abstraction found in functional languages can be extended with new operations, but adding new data variants is difficult. The traditional object-oriented approach to data abstraction facilitates adding new data variants (classes), while adding new operations is more difficult.

To address the modularity limitations of Programming Languages, several different approaches have been proposed in the past. Existing approaches can be broadly divided into two categories: *syntactic* or *semantic* modularization techniques. Syntactic modularization techniques are quite popular in practice, due to their simplicity of implementation and use. Examples include many tools for developing Feature-Oriented Software-Product Lines (SPLs) [1, 28], some Language Workbenches [19], or extensible parser generators [24, 25, 35, 47]. Most syntactic approaches employ textual composition techniques such as *superimposition* [1] to enable the development modular program features. As Kastner et. al [28] note, a typical drawback of feature-oriented SPL implementations, which more generally applies to syntactic modularity approaches, is that such "*implementation mechanisms lack proper interfaces and support neither modular type checking nor separate compilation*".

Semantic modularization techniques go one step further in terms of modularity, and also enable components or features to be modularly type-checked and separately compiled. Modular type-checking and separate compilation are desirable properties to have from a software engineering point-of-view. Modular type-checking can report errors earlier and in terms of the modular code programmers have written in the first place. Separate compilation avoids global compilation steps, which can be very costly. Furthermore semantic modularization enables the composition of compiled binaries as well as ensuring the type-safety of the code composed of multiple components. Examples of semantic modularization techniques include various approaches to *family polymorphism* [16], *virtual classes* [17], as well as various techniques for solving the Expression Problem [10, 34, 40, 45]. Semantic modularization techniques are less widely used in practice than syntactic techniques. This is partly due to the perceived need for more sophisticated type systems, which are not available in mainstream languages and may require more knowledge from users. However, recently, several lightweight modularization techniques have been shown to work in mainstream programming languages like Java or Scala. Object Algebras [10] are one such technique, which works in Java-like languages and uses simple generics only.

So far research on semantic modularization techniques has focused on operations that *traverse* or *process* extensible data structures, such as ASTs. Indeed many documented applications of semantic modularization techniques focus on modularizing various aspects of PL implementations. However, as far as we know, there is little work on operations that build/produce ASTs. In particular the problem of how to

*PL'17, New York, NY, USA*
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

modularize parsing has not been studied in semantic modularization approaches. Techniques such as NOA [24] employ a syntactic modularity approach for parsing in combination with a semantic modularity approach for defining operations that traverse or process ASTs. This is a shame because parsing is a fundamental part of PL implementations, and it ought to be made semantically modular as well, so that the full benefits of semantic modularity apply.

This paper presents a technique for doing semantically modular parsing. That is, our approach not only allows complete parsers to be built out of modular parsing components, but also enables those parsing components to be *modularly type-checked* and *separately compiled*. Developing techniques for modular parsing is not without challenges. In developing our techniques we encountered two different classes of challenges: algorithmic challenges; and typing/reuse challenges.

***Algorithmic Challenges*** A first challenge was to do with the parsing algorithms themselves, since they were usually not designed with extensibility in mind. The most widely used tools for parsing are parser generators, but they mostly require full information about the grammar to generate parsing code. Moreover, actions associated with grammar productions are typically only type-checked after the parser has been generated. Both problems go against our goals of semantic modularity.

An alternative to parser generators are *parser combinators* [7, 43]. At a first look, parser combinators seem very suitable for our purpose. Each parser combinator is represented by a piece of code directly in the programming language. Thus, in a statically typed programming language, such code is statically type-checked. However many techniques regularly employed by parser combinators cause difficulties in a modular setting. In particular, many parser combinator approaches (including Parsec [29]) routinely use *left-recursion elimination*, *priority-based matching*, and *avoid backtracking* as much as possible. All of these are problematic in a modular setting as illustrated in Section 2.1.

To address such algorithmic challenges, we propose a methodology for implementing modular parsers built on top of an existing Packrat [20] parsing library for Scala [18]. Such a library directly supports left-recursion, memoization, and a *longest-match composition* operator. We will see some examples in Section 2.2.

***Typing and Reusability Challenges*** The second class of challenges was problems related to modularity, reusability and typing of parsing code. An immediate concern is how to extend a parser for an existing language or, more generally, how to compose parsing code for two languages. It turns out that OO mechanisms that provide some form of multiple inheritance, such as traits/mixins [5, 37], are very handy for this purpose. Essentially, traits/mixins can act as modules for the parsing code of different languages. This enables an approach where ASTs can be modelled using standard OO techniques such as the Composite pattern, while retaining the possibility of adding new language constructs. Section 3 gives the details of this approach.

Our ultimate goal is to allow for full extensibility: it should be possible to modularly add not only new language constructs, but also new operations. To accomplish this goal one final tweak on our technique is to employ Object Algebras to allow fully extensible ASTs. Thus a combination of Packrat parsing, multiple inheritance and Object Algebras enables a solution for semantically modular parsing. Section 4 gives the details of the complete approach.

To evaluate our approach we conduct a case study based on the "Types and Programming Languages" (TAPL) interpreters. The case study shows that our approach is effective at reusing parsing code from existing interpreters, and the total parsing code is 69% shorter than an existing code base using non-modular parsing code[1].

In summary our contributions are:

- **A Technique for Modular Parsing:** We present a technique that allows the development of semantically modular parsers. The technique relies on the combination of Packrat parsing, multiple inheritance and Object Algebras.
- **A Parsing Technique for OO ASTs:** A simplified version of our technique also enables parsing OO-style ASTs, where new language constructs can be easily added.
- **A Methodology for Writing Modular Parsers:** We identify possible pitfalls using parser combinators. To avoid such pitfalls, we propose guidelines for writing parsing code using *left-recursion* and *longest-match composition*.
- **TAPL case study:** We conduct a case study with 18 interpreters from the TAPL book. The case study shows the effectiveness of modular parsing in terms of reuse. The TAPL case study is available online at:
  https://github.com/lihuanglx/modular-parsing

All the code in the paper and the case study is written in Scala, since its concise and elegant syntax is good for presentation. Other languages that support some form of multiple inheritance (including C++ or Java 8 with default methods [23]) could in principle be used.

## 2 Packrat Parsing for Modularity

This section discusses the algorithmic challenges introduced by modular parsing and argues that Packrat parser combinators [20] are suitable to address them. The algorithmic challenges are important because they rule out various common techniques used by non-modular code using parser combinators. To avoid pitfalls related to those algorithmic challenges, we propose the following methodology:

- **Modular parsers should support left-recursion.**
- **Modular parsers should use a longest match composition operator.**

[1]https://github.com/ilya-klyuchnikov/tapl-scala/

Moreover, the underlying parsing formalism should make *backtracking cheap*, due to its pervasiveness in modular parsing. Although we chose Packrat parsing, any other parsing formalism that provides similar features should be ok.

### 2.1 Algorithmic Challenges of Modularity

For the goal of modular parsing, parser combinators seem suitable because they are naturally modular for parser composition, but also they ensure type safety. Unfortunately many parser combinators have important limitations. In particular, several parser combinators including the famous Parsec [29] library, require programmers to manually do *left-recursion elimination*, *longest match composition*, and require significant amounts of *backtracking*. All of those are problematic in a modular setting.

***Left-Recursion Elimination*** The top-down, recursive descent parsing strategy adopted by those parser combinator libraries cannot support left-recursive grammars directly. For instance, we start with a simple arithmetic language containing only integers and subtractions. The grammar with concrete syntax and part of the parsing code in Parsec are presented below:

⟨*expr*⟩ ::= ⟨*int*⟩
  | ⟨*expr*⟩ '-' ⟨*int*⟩

```
parseExpr =
  parseSub <|> parseInt
parseSub = do
  e <- parseExpr ...
```

Such a left-recursive implementation will cause an infinite loop, since `parseExpr` and `parseSub` call each other and never stop. A common solution is to rewrite the grammar into an equivalent but non-left-recursive one, called left-recursion elimination:

⟨*expr*⟩ ::= ⟨*int*⟩ ⟨*expr'*⟩

⟨*expr'*⟩ ::= ⟨*empty*⟩
  | '-' ⟨*int*⟩ ⟨*expr'*⟩

After left-recursion elimination, the structure of grammar is changed, as well as its corresponding parser. In a modular setting, it is possible but unnecessarily complicated to analyse the grammar and rewrite it when doing extensions. Anticipating that every non-terminal has left-recursive rules is helpful for extensibility but overkill, since it is inconvenient and introduces extra complexity for representation of grammars and implementation of parsers.

Another issue of left-recursion elimination is that it requires extra bookkeeping work to retain the original semantics. For example, the expression $1 - 2 - 3$ is parsed as $(1-2)-3$ in the left-recursive grammar, but after rewrite the information of left-associativity is lost. The parse tree must be transformed to recover the correct syntactic structure.

***Longest Match Composition*** Another problematic issue in parser combinator libraries is the need for manually prioritizing/ordering alternatives in a grammar. Consider the grammar:

⟨*expr*⟩ ::= ⟨*int*⟩
  | ⟨*int*⟩ '+' ⟨*expr*⟩

In Parsec, for instance, the parser `"parseInt <|> parseAdd "` will only parse the input `"1 + 2"` to `"1"`, as `parseInt` successfully parses `"1"` and terminates parsing.

Traditional alternative composition will only find the first parser that succeeds on a prefix of the input, even if subsequent parsers may parse the whole input. In contrast to the previous parser, `"parseAdd <|> parseInt"` works as expected with because the two cases are swapped. In this case, reordering the alternatives ensures that the *longest match* is picked among the possible results. However, manual reordering for the longest match is inconvenient, and worst still, it is essentially non-modular. When the grammar is extended with new rules, programmers should *manually* adjust the order of parsers, by rewriting previously written code.

***Backtracking*** The need for backtracking can also be problematic in a modular setting. Consider a grammar that includes `"import..from"`, and is extended with an `"import.. as"` case:

⟨*stmt*⟩ ::= '`import`' ⟨*ident*⟩ '`from`' ⟨*ident*⟩
  | ...
  | '`import`' ⟨*ident*⟩ '`as`' ⟨*ident*⟩

Since the two cases share a common prefix, when the former fails, we must backtrack to the beginning. For example, the choice combinator in Parsec only tries the second alternative if the first fails without any token consumption. We have to use `try` for explicit backtracking.

```
oldParser = parseImpFrom <|> ...
newParser = try parseImpFrom <|> ... <|> parseImpAs
```

Similarly, this violates a modular setting because it also requires a global view of the full grammar. Hence the worst case where all alternatives may share common prefixes with future cases should always be anticipated. Therefore we need to backtrack for all the branches. To avoid failures in the future, we have to add `try` everywhere. However this results in the worst-case exponential time complexity.

### 2.2 Packrat Parsing

Fortunately, some more advanced parsing techniques such as Packrat parsing [20] have been developed to address limitations of simple parser combinators. Packrat parsing uses memoization to record the result of applying each parser at each position of the input, so that repeated computation is eliminated. Moreover, it supports both direct left-recursion and (in theory) indirect left-recursion [46]. All of these properties are very suitable for modularity, thus we decided to use Packrat parsers as the underlying parsing technique for modular parsing. Scala has a standard parser combinator library[2] [32] for Packrat parsers. The library provides a number of parser combinators, including the longest match alternative combinator.

[2] https://github.com/scala/scala-parser-combinators

```
import util.parsing.combinator.syntactical.
    StandardTokenParsers
import util.parsing.combinator.PackratParsers

object Code extends StandardTokenParsers
  with PackratParsers {
  type Parser[E] = PackratParser[E]
  def parse[E](p: Parser[E]): String => E = in => {
    val t = phrase(p)(new lexical.Scanner(in))
    t.getOrElse(sys.error(t.toString))
  }
  // Any Scala code in the paper comes here
}
```

**Figure 1.** Helper object for code demonstration in this paper.

***Code Demonstration*** For more concise demonstration, we assume that all the Scala code in the rest of this paper are in the object `Code`, as shown in Figure 1. It extends traits `StandardTokenParsers` and `PackratParsers` from the Scala parser combinator library. Furthermore, we will use `Parser` as a type synonym for `PackratParser` and a generic `parse` function for testing.

***Parsing a Simple Arithmetic Language*** Suppose we want to parse a simple language with literals and additions. The concrete syntax is:

$$\begin{array}{lcl}\langle expr \rangle & ::= & \langle int \rangle \\ & | & \langle expr \rangle \text{ `+' } \langle expr \rangle\end{array}$$

It is straightforward to model the abstract syntax by classes. The ASTs support pretty-printing via the `print` method.

```
trait Expr { def print: String }
class Lit(x: Int) extends Expr {
  def print = x.toString
}
class Add(e1: Expr, e2: Expr) extends Expr {
  def print = "(" + e1.print + "+" + e2.print + ")"
}
```

Then we write corresponding parsers for all cases. Note that a parser has type `Parser[E]` for some `E`, which indicates the type of results it produces.

```
trait AParser {
  lexical.delimiters += "+"
  val pLit: Parser[Expr] = numericLit ^^
    { x => new Lit(x.toInt) }
  val pAdd: Parser[Expr] = pExpr ~ ("+" ~> pExpr) ^^
    { case e1 ~ e2 => new Add(e1, e2) }
  def pExpr: Parser[Expr] = pLit ||| pAdd
}
```

In the trait `AParser`, `lexical` is used for lexing. `pLit` parses an integer for the literal case. `pAdd` handles the addition case and creates an object of `Add`. It parses two sub-expressions by calling `pExpr` recursively. Finally `pExpr` composes `pLit` and `pAdd` using the longest match alternative combinator `|||`. Table 1 shows common parser combinators from the library.

<table>
<tr><td><code>def ~[U](q: =>Parser[U]): Parser[~[T, U]]</code><br>- A parser combinator for sequential composition.</td></tr>
<tr><td><code>def ^^[U](f: (T)=>U): Parser[U]</code><br>- A parser combinator for function application.</td></tr>
<tr><td><code>def <~[U](q: =>Parser[U]): Parser[T]</code><br>- A parser combinator for sequential composition which keeps only the left result.</td></tr>
<tr><td><code>def ~>[U](q: =>Parser[U]): Parser[U]</code><br>- A parser combinator for sequential composition which keeps only the right result.</td></tr>
<tr><td><code>def ident: Parser[String]</code><br>- A parser which matches an identifier.</td></tr>
<tr><td><code>def numericLit: Parser[String]</code><br>- A parser which matches a numeric literal.</td></tr>
<tr><td><code>def |[U >:T](q: =>Parser[U]): Parser[U]</code><br>- A parser combinator for alternative composition.</td></tr>
<tr><td><code>def |||[U >:T](q0: =>Parser[U]): Parser[U]</code><br>- A parser combinator for alternative with longest match composition.</td></tr>
</table>

**Table 1.** Common combinators from the Scala standard parser combinator library.

It is worth mentioning that the left-recursive grammar above is well supported without extra code. The longest match composition is also employed by using the combinator `|||`. Furthermore, the parser does not suffer from the backtracking problem, as the memoization technique of Packrat parsing guarantees reasonable efficiency.

The code below demonstrates how to parse a valid expression `1 + 2` using our parser.

```
val p = new AParser {}
val r = parse(p.pExpr)("1 + 2").print // "(1+2)"
```

## 3 OO AST Parsing with Multiple Inheritance

Before we address the problem of full modular parsing, we first address a simpler problem: how to parse Object-Oriented ASTs. To solve this problem we employ multiple inheritance, which is supported in Scala via traits.

Part of the modular parsing problem is how to obtain an extensible parser. It is natural to make use of OO ASTs because adding new data constructs is cheap for them. Hence we have used OO traits and inheritance to represent the AST in the last section. Furthermore, we would like to write extensible parsing code on extensions of a grammar. That is to say, new extensions would not require modifying the existing code, and we can even reuse the old code.

To illustrate such extensibility, we continue with the old example, and introduce variables as a new case. It is easy to extend the corresponding OO AST together with its parser in a modular way:

```
class Var(x: String) extends Expr {
  def print = x
}
```

Here one may quickly define a new parser `pVar` in `AParser` for variables, and parse new expressions with "`pExpr ||| pVar`". Unfortunately, even "`1 + x`" cannot be parsed, which is obviously valid in the new grammar. The reason is that `pAdd` makes two recursive calls to `pExpr` for parsing subexpressions, whereas the newly added `pVar` is not observed, unless we replace all the occurrences of `pExpr` with "`pExpr ||| pVar`". Yet modifying existing code breaks semantic modularity.

***Overriding for Extensibility*** It is actually quite simple to let `pExpr` cover the newly extended case without modifying existing code. Method overriding is a standard feature which often comes with inheritance, and it allows us to redefine an inherited method, such as `pExpr`. We can build the new parser which correctly parses "`1 + x`" through overriding:

```
trait VarParser extends AParser {
  val pVar: Parser[Expr] = ident ^^ (new Var(_))
  override def pExpr: Parser[Expr] =
    super.pExpr ||| pVar
}
val p = new VarParser {}
val r = parse(p.pExpr)("1 + x").print // "(1+x)"
```

Now `VarParser` successfully represents the parser for the extended language, because Scala uses dynamic dispatch for method overriding in inheritance. When the input "`1 + x`" is fed to the parser `this.pExpr`, it firstly delegates the work to `super.pExpr`, which parses literals and additions. However, the recursive call `pExpr` in `pAdd` actually refers to `this.pExpr` again due to dynamic dispatch, and it covers the variable case. Similarly, all recursive calls can be updated to include new extensions if needed.

***Independent Extensibility*** A nice feature of Scala is its support for the linearized-style multiple inheritance on traits [18]. This can be very helpful when composing several languages, and to achieve independent extensibility [34]. Suppose now we want to compose the parsers for expressions from predefined languages `LanguageA` and `LanguageB` using alternative. The new parser can be built by inheriting both parsers at the same time:

```
trait LanguageA {...}
trait LanguageB {...}
trait LanguageC extends LanguageA with LanguageB {
  override def pExpr = super[LanguageA].pExpr |||
      super[LanguageB].pExpr
}
```

The **`super`**`[T].x` syntax in Scala, so called *static super reference*, refers to the type or method `x` in the parent trait `T`. Under multiple inheritance, it can be used to distinguish the methods of the same name. Therefore in the new parser, we use **`super`** to specify which `pExpr` we are referring to.

***Conflicts and/or Ambiguity*** In a modular setting, conflicts and ambiguity could be introduced to the grammar. In that case, the help parser combinators can offer is quite restricted. Yet users can override those problematic methods to resolve such conflicts, and rely on dynamic dispatch. We will discuss it in Section 5.2.

As demonstrated, inheritance with method overriding is the key technique to obtain semantic modularity. It enables type-safe code reuse and separate compilation for parsing OO style ASTs.

## 4 Full Extensibility with Object Algebras

The inheritance-based approach allows building extensible parsers, based on an OO class hierarchy. Nevertheless, the addition of new operations over ASTs is problematic using traditional OO ASTs. In this section, we show how to support both forms of extensibility on ASTs (easy addition of language constructs, and easy addition of operations) using Object Algebras [10].

### 4.1 Problem with Traditional OO ASTs

The Expression Problem [44] illustrates the difficulty of extending data structures or ASTs in two dimensions. In brief, it is hard to add new operations with traditional OO ASTs. In the last section we have seen a language that supports pretty-printing (in `Expr`). To modularly add an operation like collecting free variables, one attempt would be extending `Expr` with the new operation to obtain a new abstract type for ASTs:

```
trait NewExpr extends Expr { def free: Set[String] }
```

Then all classes representing language constructs could be extended to implement the operation. A first well-known problem is that such approach is problematic in terms of type-safety (but see recent work by Wang and Oliveira [45], which shows a technique that is type-safe in many cases). More importantly, a second problem is that even if that approach would work, the parsing code in `VarParser` is no longer reusable! The types `Expr, Lit, Add`, and so on, are all old types without the free variables operation. To match the new ASTs, we have to substitute `NewExpr` for `Expr` (the same for `Lit`, `Add`, ...). This requires either code modification or type casts. The goal of semantic modularity motivates us to find a different approach for building ASTs.

### 4.2 Object Algebras

Fortunately, Object Algebras [10] enable us to solve this problem. They capture a design pattern that addresses the Expression Problem, achieving two dimensions of extensibility (language constructs and operations) in a modular and type-safe way. The definition of data structures is separated from their behaviours, and future extensions on both dimensions no longer require existing code to be modified, supporting separate compilation.

Using Object Algebras in Scala, ASTs as recursive data structures are defined by traits, where each constructor corresponds to an abstract method inside. Essentially, Object

Algebras generalize the Abstract Factory pattern [22], and promote the use of factory methods, instead of constructors, for instantiating objects. The example from Section 2.2 is used here again for illustration. At first the language only supports literals and additions:

```
trait Alg[E] {
  def lit(n: Int): E
  def add(e1: E, e2: E): E
}
```

Here `Alg` is called an *Object Algebra interface*, parameterized by the type `E`, which abstracts over the concrete type of the AST.

***Adding New Operations*** To realize an operation on expressions, we simply instantiate the type parameter by a concrete type and provides implementations for all cases. Below is an example of pretty-printing:

```
trait Print extends Alg[String] {
  def lit(n: Int) = n.toString
  def add(e1: String, e2: String) =
    "(" + e1 + " + " + e2 + ")"
}
```

Here `Print` is called an *Object Algebra*. It traverses an expression bottom-up, and returns a string as the result. One can also define an evaluation operation as a new trait that extends `Alg[Int]`. Hence adding new operations is modular. We omit that code due to space reasons.

***Adding New AST Constructs*** Furthermore, new language constructs can be added by extending `Alg` and adding new cases only. Now we extend the language with variables. A new Object Algebra interface `VarAlg` is defined as follows:

```
trait VarAlg[E] extends Alg[E] {
  def varE(x: String): E
}
```

Now pretty-printing on the new language can be realized without modifying existing code:

```
trait VarPrint extends VarAlg[String] with Print {
  def varE(x: String) = x
}
```

An observation is that only the new case is implemented for pretty-printing, and the others have been inherited. Thus existing code was reused and was not modified!

To create an expression representing `1 + x`, a generic method is defined as follows:

```
def makeExp[E](alg: VarAlg[E]): E =
  alg.add(alg.lit(1), alg.varE("x"))
```

Note how the construction of the abstract syntax happens through the use of factory methods, instead of constructors. To pretty-print the expression, the code `"makeExp(new VarPrint {})"` results in `"(1 + x)"` as expected.

```
trait OAParser[E] {
  lexical.delimiters += "+"
  val alg: Alg[E]
  val pLit: Parser[E] = numericLit ^^
    { x => alg.lit(x.toInt) }
  val pAdd: Parser[E] = pE ~ ("+" ~> pE) ^^
    { case e1 ~ e2 => alg.add(e1, e2) }
  val pExpr: Parser[E] = pLit ||| pAdd
  val pE: Parser[E] = pExpr
}
```

**Figure 2.** Pattern of modular parsing using Object Algebras.

### 4.3 Parsing with Object Algebras

Parsing produces ASTs as the result. When Object Algebras are used to build ASTs, an Object Algebra containing the constructor/factory methods has to be used by the parsing function. Thus, a first attempt at defining the parser for the small arithmetic language is:

```
trait Attempt[E] {
  lexical.delimiters += "+"
  val pLit: Alg[E] => Parser[E] = alg =>
    numericLit ^^ { x => alg.lit(x.toInt) }
  val pAdd: Alg[E] => Parser[E] = alg =>
    pExpr(alg) ~ ("+" ~> pExpr(alg)) ^^
      { case e1 ~ e2 => alg.add(e1, e2) }
  val pExpr: Alg[E] => Parser[E] = alg =>
    pLit(alg) ||| pAdd(alg)
}
```

Such a parser looks fine, but it is not extensible. For example, we have demonstrated in Section 3 that method overriding is essential to update `pExpr` for an extended syntax. However, trying to do a similar method overriding for `pExpr` would require a type `VarAlg[E] => Parser[E]`, which is a *supertype* of the old type `Alg[E] => Parser[E]`, since the *extended* Object Algebra interface appears in *contravariant* position. This violates overriding in Scala.

***A Solution*** A solution to this problem is to declare a field of Object Algebra interface in the parser. Figure 2 shows the code of true modular parser, whose methods can be overridden for future extension.

That is precisely the pattern that we advocate for modular parsing. One important remark is we introduce `pE` for recursive calls. The reason why we use it as an extra and seemingly redundant field, is due to a subtle issue caused by Scala language and its parser combinator library. There is a restriction of `super` keyword in Scala that `super` can only use methods defined by keyword `def`, but cannot access fields defined by `val`, while the parser combinator library suggests using `val` to define parsers, especially for left-recursive ones. Our workaround is that we use different synonyms for `pE` in different traits, so that we can directly distinguish them by names without using `super`.

***Extensions*** Now let's try on the variables extension:

```
trait VarOAParser[E] extends OAParser[E] {
  override val alg: VarAlg[E]
  val pVar: Parser[E] = ident ^^ alg.varE
  val pVarExpr: Parser[E] = pExpr ||| pVar
  override val pE: Parser[E] = pVarExpr
}
```

The type of the Object Algebra field `alg` is first refined to `VarAlg[E]`, to allow calling the additional factory method for variables. Unlike the previous attempt, such a type-refinement is allowed. Now, the code for parsing variables (`pVar`) can call `alg.varE`. The following code illustrates how to use the parser from a client's perspective:

```
val p = new VarOAParser[String] {
  override val alg = new VarPrint {}
}
val r = parse(p.pE)("1 + x") // "(1 + x)"
```

In the client code above, we pick the pretty-printing algebra `VarPrint` to initialize the `alg` field, but any other Object Algebra that implements `VarAlg` would work. With an instance of `VarOAParser` in hand, we can call `pE` to obtain the parser to feed to the `parse` method. Such a pattern provides modular parsing as expected.

Note that, similar to the approach in Section 3, independent extensibility is also supported via multiple trait inheritance. Since it is achieved using essentially the same technique as in Section 3, we omit the code here.

# 5 More Features

The use of inheritance-based approach and Object Algebras enables us to build modular parsers, which are able to evolve with syntax together. This section explores more interesting features, including parsing multi-sorted syntax, overriding existing parsing rules, language components for abstracting language features, and alternative techniques under the whole framework.

## 5.1 Parsing Multi-Sorted Syntax

Using Object Algebras, it is easy to model multi-sorted languages. If the syntax has multiple sorts, we can distinguish them by different type parameters. For instance, we extend the expression language from the end of Section 4, with a primitive type `int` type and typed lambda abstractions:

⟨*type*⟩ ::= '`int`'

⟨*expr*⟩ ::= ...
  | '`\`' ⟨*ident*⟩ '`:`' ⟨*type*⟩ '`.`' ⟨*expr*⟩

The code below illustrates the corresponding Scala code that extends the Object Algebra interface, pretty-printing operation and parser.

```
trait LamAlg[E, T] extends VarAlg[E] {
  def intT(): T
  def lam(x: String, t: T, e: E): E
}
trait LamOAParser[E, T] extends VarOAParser[E] {
  lexical.reserved += "int"
  lexical.delimiters += ("->", "\\", ":", ".")
  override val alg: LamAlg[E, T]
  val pIntT: Parser[T] = "int" ^^ { _ => alg.intT }
  val pTypedLamT: Parser[T] = pIntT
  val pLam: Parser[E] =
    ("\\" ~> ident) ~ (":" ~> pT) ~ ("." ~> pE) ^^
      { case x ~ t ~ e => alg.lam(x, t, e) }
  val pTypedLamE: Parser[E] = pVarExpr ||| pLam
  val pT: Parser[T] = pTypedLamT
  override val pE: Parser[E] = pTypedLamE
}
```

We use two type parameters `E` and `T` for expressions and types. The type system guarantees that invalid terms such as `int + int` will be rejected. Besides lexing, the trait `LamOAParser` also introduces parsers for types, and the new case for expressions. We use `pTypedLamT` and `pTypedLamE` as copies of current `pT` and `pE`, due to the issue with `super` in Scala (see discussion in Section 4.3). `pT` and `pE` are used for recursion.

## 5.2 Overriding Existing Rules

As many syntactically extensible parsers, our approach also supports modifying part of existing parsers, including updating or eliminating existing rules, but in a type-safe way. This can be useful in many situations, for instance when conflicts or ambiguities arise upon composing languages. As an illustration, suppose we have an untyped lambda abstraction case in a base parser, defined as a value:

```
val pLam: Parser[E] =
  ("\\" ~> ident) ~ ("." ~> pE) ^^ ...
```

Here `pLam` parses a lambda symbol, an identifier, a dot and an expression in sequence. Then we want to replace the untyped lambda abstractions by typed lambdas. With inheritance and method overriding, it is easy to only change the implementation of `pLam` in the extended parser. Due to dynamic dispatch, our new implementation of lambdas will be different without affecting the other parts of the parser.

```
override val pLam: Parser[E] =
  ("\\" ~> ident) ~ (":" ~> pT) ~ ("." ~> pE) ^^ ...
```

One can even "eliminate" a production rule in the extension, by overriding it with a failure parser. The lexer can also be updated, since keywords and delimiters are represented by sets of strings.

## 5.3 Language Components

Modular parsing not only enables us to build a corresponding parser which evolves with the language together, but also allows us to abstract language features as reusable, independent components. Generally, a language feature includes related abstract syntax, methods to *build* the syntax (parsing), and methods to *process* the syntax (evaluation, pretty-printing, etc.). From this perspective, not only one language, but many languages can be developed in a modular way, with common language features reused.

Instead of designing and building a language from scratch, we can easily add a new feature by reusing the corresponding language component. For example, if a language is composed from a component of boolean expressions, including if-then-else, it immediately knows how to parse, traverse, and pretty-print the if-then-else structure. Grouping language features in this way can be very useful for rapid development of DSLs.

For implementation, a language component is represented by a Scala object, and it consists of three parts: Object Algebra interface, parser, and Object Algebras.

- **Object Algebra interface:** defined as a trait for the abstract syntax. The type parameters represent multiple sorts of syntax, and methods are constructs.
- **Parser:** corresponding parser of the abstract syntax, written in a modular way as we demonstrated before.
- **Object Algebras (optional):** concrete operations on ASTs, such as pretty-printing.

We take the example in Section 4.3 again. It can be defined as a language component `VarExpr`. For space reasons we omit some detailed code.

```
object VarExpr {
  trait Alg[E] { // Abstract syntax
    def lit(n: Int): E
    ...
  }
  trait Parse[E] { ... } // Parser
  trait Print extends Alg[String] {
    ... // Pretty-printer
  }
}
```

For the extension of types and lambda abstractions in Section 5.1, instead of inheriting from the previous language directly, we can define it as another independent language component `TypedLam`.

```
object TypedLam {
  trait Alg[E, T] { // Abstract syntax
    def intT(): T
    ...
  }
  trait Parse[E, T] { ... } // Parser
  trait Print extends Alg[String, String] {
    ... // Pretty-printer
  }
}
```

The code below shows how we merge those two components together to obtain the language we want. Furthermore, the new language is still a modular component ready for future composition. In that case modularity is realized over higher-order hierarchies.

```
object VarLamExpr {
  trait Alg[E, T] extends VarExpr.Alg[E]
    with TypedLam.Alg[E, T]
  trait Parse[E, T] extends VarExpr.Parse[E]
    with TypedLam.Parse[E, T] {
    override val alg: Alg[E, T]
    override val pE: Parser[E] = ...
    ...
  }
  trait Print extends VarExpr.Print
    with TypedLam.Print
}
```

The only drawback is that the glue code of composition appears to be boilerplate. As shown above, we are combining ASTs, parsers and pretty-printers of `VarExpr` and `TypedLam` respectively. Such a pattern refers to *family polymorphism* [16] which is unfortunately not fully supported in Scala, since nested classes/traits have to be manually composed.

### 5.4 Alternative Techniques

Our prototype uses Packrat parsing as the underlying parsing technique, OO inheritance for composing and extending parsers, and Object Algebras for parsing extensible ASTs. Yet such a framework is itself flexible and modular, because those techniques can have alternatives. For example, as we mentioned before, any parsing library that resolves the algorithmic challenges in modular parsing can work well. Regarding OO inheritance for the extensibility, an alternative approach, called *open recursion* [8] can be used in other languages, by introducing explicit "self-reference" parameters for the recursion. Furthermore, besides Object Algebras, *Data types à la carte* (DTC) [38] and the Cake pattern [34] also support extensible data structures. For the goal of modular parsing a custom combination of those alternatives can be adopted.

## 6 Case Study

To demonstrate the utility of our modular parsing approach, we implemented parsers for the first 18 calculi [3] from the *Types and Programming Languages* (TAPL) [36] book. We compared our implementation with a non-modular implementation available online, which is also written in Scala and uses the same Packrat parsing library. We counted source lines of code (SLOC) and measured execution time for both implementations. The result suggests that our implementation saves 69% code comparing with that non-modular one, but there is a 43% slowdown due to code modularity.

### 6.1 Implementation

TAPL introduces several calculi from simple to complex, by gradually adding new features to syntax. These calculi are suitable for our case study for mainly two reasons. Firstly, they capture many of the language features required in realistic programming languages, such as lambdas, records and polymorphism. Secondly, the evolution of calculi in the book reveals the advantages of modular representation of abstract

[3]There are some more calculi in the book, but they are either not ported by the implementation we compare with, or just repeats the syntax of former ones.

syntax and modular parsing, which is the key functionality of our approach. By extracting common components from those calculi and reusing them, we obtain considerably code reuse as shown later.

We extract reusable components from all the calculi using the pattern demonstrated in Section 5.3. Each component, which may contain several syntactical structures, represents a certain feature. They are combined together as needed to build a calculus. For example, the calculus `Untyped` in our case study, representing the famous untyped lambda calculus, consists of component `VarApp` (for variables and applications) and component `UntypedAbs` (for untyped lambdas).

Figure 3 shows the dependency of all the components and calculi in our case study. Grey boxes are calculi and white boxes are components. An arrow starting from box A to box B denotes that B includes and thus reuses A.

Each component or language is represented by a Scala object which includes `Alg` for the abstract syntax, `Print` for pretty-printing, and `Parse` for parsing. Since calculi and components have similar signatures, each calculus can also be extended and reused directly. For example, calculus `FullRef` extends from calculus `FullSimple`.

### 6.2 Comparison

We compared our implementation (named $\texttt{Mod}_{\texttt{OA}}$) with an implementation available online[4] (named `NonMod`). `NonMod` is suitable for comparison, because it is also written in Scala using the same parser combinator library. `NonMod` implements parsers 18 calculi in TAPL in a non-modular way. Thus `NonMod` is not able to reuse existing code when those calculi share common features. $\texttt{Mod}_{\texttt{OA}}$ implements the same 18 calculi, but reuse is possible due to modularity.

The comparison is made from two aspects. First, we want to discover the amount of code reuse using our modular parsing approach. For this purpose, we measured source lines of code (SLOC) of two implementations. Second, we are interested to assess the performance penalty caused by modularity. Thus we compared the execution time of parsing random expressions between two implementations.

***Standard of Comparison*** In the SLOC comparison, all blank lines and comments are excluded, and we formatted the code of both implementations to guarantee that the length of each line does not exceed 120 characters. Furthermore, because `NonMod` has extra code such as semantics, we removed all irrelevant code and only keep abstract syntax definition, parser and pretty-printer for each calculus, to ensure a fair comparison.

For the comparison of execution time, we built a generator to randomly generate valid expressions for each calculus, according to its syntax. These expressions are written to test files, one file per calculus. Each test file consists of 500 expressions randomly generated, and the size of test files varies from 20KB to 100KB. We run the corresponding parser to parse the file and the pretty-printer to print the result. The average execution time of 5 runs excluding reading input file was calculated, in milliseconds.

***Comparison Results*** Table 2 shows results of the comparison. Let us only check $\texttt{Mod}_{\texttt{OA}}$ and `NonMod` for now. The overall result is that 69.2% of code is reduced using our approach, and our implementation is 42.7% slower.

The good SLOC result is because of that the code of common language features are reused many times in the whole case study. We can see that in the first two calculi `Arith` and `Untyped` we are not better than `NonMod`, because in such two cases we do not reuse anything. However in the following 16 calculi, we indeed reuse language components. In particular, the calculi `EquiRec` and some others are only 22 lines in our implementation, because we only compose existing code.

To discover the reasons of slower execution time, we made experiments on two possible factors, which are Object Algebras and the longest match alternative combinator. We use Object Algebras for ASTs and the longest match alternative combinator `|||` for parsing, while `NonMod` uses case class and the ordinary alternative combinator. Therefore, we implemented two more versions. One is a modified version of our implementation, named $\texttt{Mod}_{\texttt{CLASS}}$, with Object Algebras replaced by case class for the ASTs. The other is a modified version of `NonMod`, named $\texttt{NonMod}_{|||}$, using the longest match alternative combinator instead of the ordinary one.

The right part of Table 2 suggests that the difference of running time between using Object Algebras and class is little, roughly 1%. The usage of longest match combinator slows the performance by 7%. The main reason of slower execution time may be the overall structure of the modular parsing approach, because we indeed have more intermediate function calls and method overriding. However, it is worth mentioning that because of the memoization technique of Packrat parsers, we are only constant times slower, the algorithmic complexity is still the same. Since the slowdown seems to be caused by extra method dispatching, in future work we wish to investigate techniques such as partial evaluation or meta-programming to eliminate such cost. The work by Béguet and Manohar [4] is an interesting starting point.

## 7 Related Work

Our work touches upon several topics including extensible parsing, parser combinators and extensibility techniques.

***Syntactically Extensible Parsing*** Extensible parser generators [24, 25, 35, 41, 47] are a mainstream area of modular syntax and parsing. They allow users to write modular grammars, where new non-terminals and production rules can be introduced, some can even override existing rules in the old grammar modules. For instance, *Rats!* [25] constructs its own module system for the collection of grammars, while

[4]https://github.com/ilya-klyuchnikov/tapl-scala/

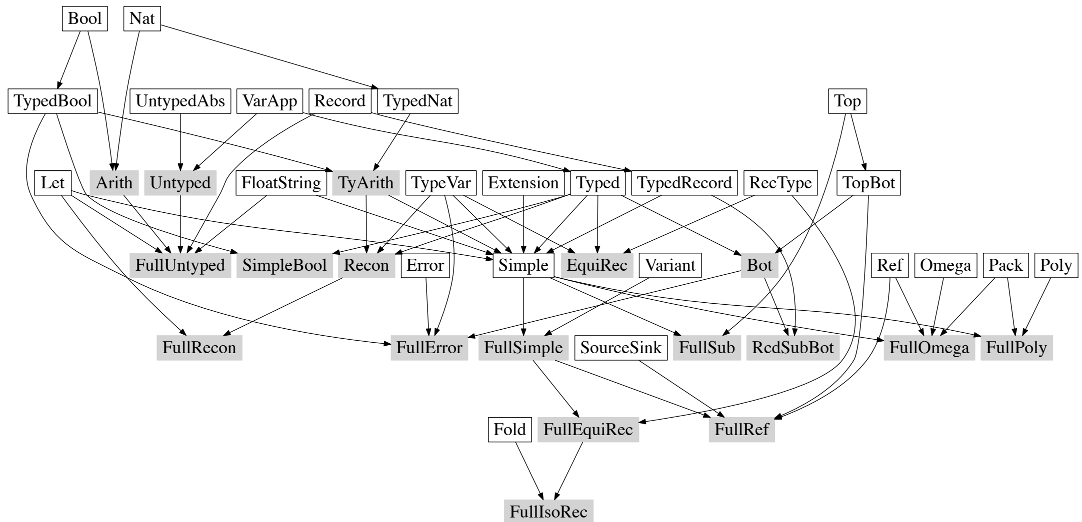


**Figure 3.** Dependency graph of all calculi and components. Grey boxes are calculi; white boxes are components.

| **Calculus Name** | **SLOC** | | | **Time (ms)** | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | NonMod | Mod$_{OA}$ | (+/-)% | NonMod | Mod$_{OA}$ | (+/-)% | NonMod$_{|||}$ | (+/-)% | Mod$_{CLASS}$ | (+/-)% |
| Arith | 77 | 77 | +0.0 | 741 | 913 | +23.2 | 793 | +7.0 | 932 | +25.8 |
| Untyped | 48 | 53 | +10.4 | 770 | 1018 | +32.2 | 821 | +6.6 | 1007 | +30.8 |
| FullUntyped | 131 | 75 | -42.7 | 1297 | 1854 | +42.9 | 1343 | +3.5 | 1767 | +36.2 |
| TyArith | 89 | 54 | -39.3 | 746 | 888 | +19.0 | 772 | +3.5 | 918 | +23.1 |
| SimpleBool | 90 | 42 | -53.3 | 1376 | 1782 | +29.5 | 1494 | +8.6 | 1824 | +32.6 |
| FullSimple | 244 | 127 | -48.0 | 1441 | 2270 | +57.5 | 1574 | +9.2 | 2226 | +54.5 |
| Bot | 87 | 48 | -44.8 | 1080 | 1287 | +19.2 | 1078 | -0.2 | 1306 | +20.9 |
| FullRef | 277 | 65 | -76.5 | 1438 | 2291 | +59.3 | 1544 | +7.4 | 2142 | +49.0 |
| FullError | 112 | 41 | -63.4 | 1410 | 1946 | +38.0 | 1524 | +8.1 | 1981 | +40.5 |
| RcdSubBot | 125 | 22 | -82.4 | 1247 | 1524 | +22.2 | 1285 | +3.0 | 1612 | +29.3 |
| FullSub | 225 | 22 | -90.2 | 1320 | 1979 | +49.9 | 1393 | +5.5 | 1899 | +43.9 |
| FullEquiRec | 250 | 36 | -85.6 | 1407 | 2200 | +56.4 | 1561 | +10.9 | 2156 | +53.2 |
| FullIsoRec | 259 | 40 | -84.6 | 1492 | 2253 | +51.0 | 1648 | +10.5 | 2236 | +49.9 |
| EquiRec | 81 | 22 | -72.8 | 994 | 1254 | +26.2 | 1048 | +5.4 | 1304 | +31.2 |
| Recon | 138 | 22 | -84.1 | 1044 | 1482 | +42.0 | 1128 | +8.0 | 1506 | +44.3 |
| FullRecon | 142 | 22 | -84.5 | 1094 | 1645 | +50.4 | 1161 | +6.1 | 1652 | +51.0 |
| FullPoly | 248 | 68 | -72.6 | 1398 | 2086 | +49.2 | 1511 | +8.1 | 2019 | +44.4 |
| FullOmega | 315 | 68 | -78.4 | 1451 | 2352 | +62.1 | 1582 | +9.0 | 2308 | +59.1 |
| **Total** | 2938 | 904 | **-69.2** | 21746 | 31024 | **+42.7** | 23260 | +7.0 | 30795 | +41.6 |

**Table 2.** Comparison of SLOC and execution time.

NOA [24] uses Java annotation to collect all information before producing an ANTLR [35] grammar and the parsing code. Those parser generators focus on the *syntactic extensibility* of grammars: they rely on whole compilation to generate a global parser, even if there is only a slight modification in the grammar. Some of those parser generators may statically check the correctness and unambiguity of grammars. In contrast, because our approach is based on parser combinators, there is no support for ambiguity checking. However, as far as we are aware, no extensible parser generators support separate compilation or modular type-checking. It is worth mentioning that in [41], users can define grammar fragments as typed Haskell values, and combine them on the fly. Later they are processed by a typed parser generator. Nevertheless this requires a lot of advanced language features, making client complex. Our approach is simple and a straightforward use of OO programming, and makes parsing code directly reusable.

Macro systems like the C preprocessor, C++ templates and Racket [39], and other meta-programming techniques are a similar area aiming at syntactic extensibility. SugarJ [15] conveniently introduces syntactic sugar for Java using library imports. Composition of syntactic sugar is easy for users, but it requires many rounds of parsing and adaption, hence significantly affects the efficiency of compilation. Since the implementation was based on SDF [26] and Stratego [42], it does not support separate compilation. Racket adopts a macro system for library-based language extensibility [39]. It uses attributed ASTs for contextual information, and extensions can be integrated in a modular way. However such modularity is not flexible enough for language unification, as the syntax is only built from extensions. Extensible compilers like JastAdd [13] and Polyglot [33] also support extensible parsing, but it is mostly done using parser generators. They focus on the extensions to a host language. Those techniques are short of type safety in a modular setting as well.

***Extensible Parsing Algorithms*** *Parse table composition* [6] is an approach where grammars are compiled to modular parse tables. Those parse tables are expressed as DFAs or NFAs, and later they can be composed by an algorithm, to provide separate compilation for parsing. The generation of parse tables can be quite expensive in terms of performance. The approach is quite different from ours, since it uses parse tables, whereas we use parser combinators. Our approach supports both *separate compilation* as well as *modular type-checking*. Moreover, the extensibility of parsing is further available at language composition and lexical level.

***Parser Combinators*** Parser combinators have become more and more popular since [7, 43]. Many parsing libraries produce recursive descent parsers by introducing functional monadic parser combinators [27]. Parsec [29] is perhaps the most popular parser combinator library in this line. It is widely used in Haskell (with various "clones" in other languages) for context-sensitive grammars with infinite lookahead. Nevertheless, Parsec users suffer from manual left-recursion elimination, high cost for backtracking and longest match composition issues, as we discussed in Section 2.1. Those limitations make Parsec (and similar parsing techniques) inadequate for modular parsing.

Some recent work on parser combinators [20, 21, 31] proposed a series of novel parsing techniques that address the issue of left-recursion. We chose Packrat parsing due to its simplicity in Scala, but in general there are alternatives to it.

***Extensibility*** Various design patterns [22] in multiple languages, have been proposed over the years to address extensibility problems, such as the Expression Problem [44]. The famous "Datatypes à la Carte" (DTC) [38] approach represents modular ASTs using co-products of every two functors. Several variants of DTC have been later proposed [2, 3, 11]. All of that work essentially covers how to traverse and consume extensible ASTs. However they do not address the problem of *modularly parsing extensible ASTs*. Only in Bahr's [3] work *unfolds* is briefly mentioned, yet it does not cover parsing.

There are also many design patterns in OO languages that achieve type-safe extensibility [9, 10, 34, 40, 45]. We chose Object Algebras [10] because the pattern is relatively lightweight and makes good use of existing OO features, such as inheritance, generics and subtyping. As seen throughout the paper, the parsing code is concise and expressive using Object Algebras. On the other hand we are unware of any work on OOP that has covered how to do modular parsing for extensible ASTs.

It is worth mentioning that Scala case classes [14] provide a near solution to the Expression Problem. Case classes can be modularly added, but they do not enforce exhaustiveness of pattern matching for extensible operations. In other words run-time pattern matching errors can happen when writing extensible code with case classes. So that full static type-safety is not ensured. Nevertheless case classes are a pragmatic approach, which is widely used in practice. Therefore, modular parsing techniques may be of value for extensible code using case classes. The approach we presented in Section 3 can readily be adapted to case classes: all that the users need to do is to use case classes, instead of standard OO classes in their code.

## 8 Conclusion

This paper presents a solution for type-safe modular parsing. Our solution not only enables parsers to evolve together with the abstract syntax, but also allows parsing code to be modularly type-checked and separately compiled.

We identify the algorithmic challenges of building modular parsers, and use standard OO techniques including inheritance and overriding for our goal. However, the extensibility issue of traditional OO ASTs motivates us to adopt Object Algebras for full extensibility and more useful features. Then language feature abstraction further enhances code reuse and modularity. The TAPL case study demonstrates that a lot of boilerplate can be reduced by modular parsing.

There are certainly some aspects that can be improved. We observed that the glue code of composition appears to be boilerplate, for which family polymorphism [16] is a potential solution. Moreover, we can possibly adopt the Shy framework [48] and algebra composition patterns [12], to improve the usage of Object Algebras. For future work, it will be interesting to see how modular parsing appears in functional programming languages, as they usually do not support subtyping or inheritance. Potentially open recursion [8] can contribute.

## References

[1] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (2009), 49–84.

[2] Patrick Bahr. 2014. Composing and decomposing data types: a closed type families implementation of data types à la carte. In *Proceedings of WGP 2014*. 71–82.

[3] Patrick Bahr and Tom Hvitved. 2011. Compositional data types. In *Proceedings of WGP@ICFP 2011*. 83–94.

[4] Eric Béguet and Manohar Jonnalagedda. 2014. Accelerating Parser Combinators with Macros. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*.

[5] Gilad Bracha and William R. Cook. 1990. Mixin-based Inheritance. In *Proceedings of OOPSLA/ECOOP 1990*. 303–311.

[6] Martin Bravenboer and Eelco Visser. 2008. Parse Table Composition. In *Proceedings of SLE 2008*. 74–94.

[7] William H. Burge. 1975. *Recursive programming techniques*. Addison-Wesley Longman, Incorporated.

[8] William R. Cook. 1989. *A Denotational Semantics of Inheritance*. Ph.D. Dissertation. Brown University.

[9] Bruno C. d. S. Oliveira. 2009. Modular Visitor Components. In *Proceedings of ECOOP 2009*. 269–293.

[10] Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses - Practical Extensibility with Object Algebras. In *Proceedings of ECOOP 2012*. 2–27.

[11] Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. 2015. Modular reifiable matching: a list-of-functors approach to two-level types. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell*. 82–93.

[12] Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *Proceedings of ECOOP 2013*. 27–51.

[13] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of OOPSLA 2007*. 1–18.

[14] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *Proceedings of ECOOP 2007*. 273–298.

[15] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proceedings of OOPSLA 2011*. 391–406.

[16] Erik Ernst. 2001. Family Polymorphism. In *Proceedings of ECOOP 2001*. 303–326.

[17] Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *Proceedings of POPL 2006*. 270–282.

[18] Martin Odersky et al. 2004. *An Overview of the Scala Programming Language*. Technical Report IC/2004/64. EPFL Lausanne, Switzerland.

[19] Sebastian Erdweg et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47.

[20] Bryan Ford. 2002. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *Proceedings of ICFP 2002*. 36–47.

[21] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Proceedings of PADL 2008*. 167–181.

[22] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

[23] Brian Goetz and Robert Field. 2012. Featherweight Defenders: A formal model for virtual extension methods in Java. *Oracle Corporation, Mar* 27 (2012), 9.

[24] Maria Gouseti, Chiel Peters, and Tijs van der Storm. 2014. Extensible language implementation with object algebras (short paper). In *Proceedings of GPCE 2014*. 25–28.

[25] Robert Grimm. 2006. Better extensibility through modular syntax. In *Proceedings of PLDI 2006*. 38–51.

[26] Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. 1989. The syntax definition formalism SDF. *ACM Sigplan Notices* 24, 11 (1989), 43–75.

[27] Graham Hutton and Erik Meijer. 1996. *Monadic parser combinators*. Technical Report NOTTCS-TR-96-4. University of Nottingham, University of Nottingham. http://eprints.nottingham.ac.uk/237/

[28] Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The road to feature modularity?. In *Proceedings of the 15th International Software Product Lines Conference, 2011*. 5.

[29] Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Technical Report UU-CS-2001-3. Department of Information and Computing Sciences, Utrecht University.

[30] M. Douglas McIlroy. 1968. Mass Produced Software Components. Report on a Conference of the Nato Science Committee. (1968).

[31] Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *Proceeding of ICFP 2011*. 189–195.

[32] Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Parser combinators in Scala. (2008).

[33] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*. 138–152.

[34] Martin Odersky and Matthias Zenger. 2005. Independently extensible solutions to the expression problem. In *Proceeding of FOOL 2015*.

[35] Terence John Parr and Russell W Quong. 1995. ANTLR: A predicated-LL(K) Parser Generator. *Softw. Pract. Exper.* 25, 7 (July 1995), 789–810.

[36] Benjamin C Pierce. 2002. *Types and programming languages*. MIT press.

[37] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *Proceedings of ECOOP 2003*. 248–274.

[38] Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 04 (2008), 423–436.

[39] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of PLDI 2011*. 132–141.

[40] Mads Torgersen. 2004. The Expression Problem Revisited. In *Proceedings of ECOOP 2004*. 123–143.

[41] Marcos Viera, Doaitse Swierstra, and Atze Dijkstra. 2012. Grammar Fragments Fly First-class. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA '12)*. Article 5, 7 pages.

[42] Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*. 357–362.

[43] Philip Wadler. 1985. How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Proceedings of Functional Programming Languages and Computer Architecture, 1985*. 113–128.

[44] Philip Wadler. 1998. The expression problem. *Java-genericity mailing list* (1998).

[45] Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The expression problem, trivially!. In *Proceedings of the 15th International Conference on Modularity*. 37–41.

[46] Alessandro Warth, James R. Douglass, and Todd D. Millstein. 2008. Packrat parsers can support left recursion. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*. 103–110.

[47] Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. 2016. Modular semantic actions. In *Proceedings of the 12th Symposium on Dynamic Languages*. 108–119.

[48] Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. 2015. Scrap your boilerplate with object algebras. In *Proceedings of OOPSLA 2015*. 127–146.