Abstract of thesis entitled

"Gems: Language Modularization, Semantically"

Submitted by

Huang LI

for the degree of Master of Philosophy
at The University of Hong Kong
in September, 2017

Great efforts are needed to create new programming languages, and to maintain and improve existing ones, since it requires both knowledge about language design and implementation, and knowledge about specific domains. Therefore, it is a desire to modularize languages as components and share common features among different languages, including syntax and semantics. Most current tools towards this goal, including many extensible compilers and language workbenches, depend on metaprogramming and code generation. They achieve only *syntactic modularity* by allowing users to write code separately. *Semantic modularity*, which requires the components to be *modularly type-checked* and *separately compiled*, is not fully available.

A part of the language modularization issue is abstracted by the *expression problem*, which requires a recursive datatype (abstract syntax tree, AST) to be extended with both new operations (semantics) and new data variants (abstract syntax structures), while retaining semantic modularity. However, some solutions to the expression problem exist only on paper and practical tools and experiments are missing. Furthermore, the expression problem only focuses on operations that *consume* such extensible ASTs, while operations that *produce* ASTs are neglected, such as parsing, the bridge between concrete and abstract syntax.

In the thesis, we present our work on modularizing the whole pipeline of languages. We build a Scala framework called Gems (Langua**ge** **M**odularization, **S**emantically), which consists of two parts: a set of techniques to modularize languages, and a metaprogramming library for practical development. Semantic modularity is guaranteed in every aspect: parsing, abstract syntax, semantics (operations on ASTs). We use techniques of *type-safe modular parsing*, and *modular external visitors* with some enhancements. Gems only requires features that have strong theoretical background in object-oriented programming, including higher-order generics, type variance, and multiple inheritance. Scala-specific features such as case classes are not needed. Moreover, Gems does not rely on metaprogramming to achieve modularity, but only uses it to generate local boilerplate code. To evaluate its utility, we conduct a case study by implementing interpreters of the first 18 languages in book *Types and Programming Languages*, and compare with a non-modular implementation.

(*337 words*)

THE UNIVERSITY OF HONG KONG

MASTER'S THESIS

Gems: Language Modularization, Semantically

Author:
Huang LI

Supervisor:
Dr. Bruno C. d. S. OLIVEIRA

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Philosophy
at The University of Hong Kong

September, 2017

Declaration

I declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Signed: ____

Huang LI

Acknowledgments

First and foremost, I would like to give my sincere gratitude to my supervisor, Dr. Bruno C. d. S. Oliveira, who gave me the chance to study the interesting topic of programming languages in HKU. I am really grateful for his wonderful supervision, valuable encouragement, and kind support during my whole study.

I would like to thank my friends in the programming language group: Weixin Zhang, Xuan Bi, Yanpeng Yang, Ningning Xie, Yanlin Wang, and Tomas Tauber. Especially, I would like to thank Haoyuan Zhang, who collaborated with me on the work of modular parsing [70] (Chapter 5 of the thesis). I am also very grateful to my other friends in CB-430: Kan Wu and Xuhui Jia, my friends in neighbour laboratories, and many other friends in Hong Kong. Thank you all for your company and help. I will never forget the happy memories in the two years.

Last but not least, I would like to thank my parents, who always give me selfless support and love since the very beginning of my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The idea of dividing a whole software program into modular components was introduced decades ago [41]. In terms of programming language development, it is a desire to modularize languages and share common features among different languages, including syntax and semantics.

This thesis presents a language modularization framework called Gems (Language Modularization, Semantically), which allows languages to be divided into reusable components and developed modularly. It consists of two parts:

- **A set of techniques to semantically modularize languages**, including parsers, abstract syntax, and semantics (operations on ASTs).
- **A metaprogramming library for practical development**, which generates boilerplate code according to the abstract syntax, for reducing effort in implementation.

Beyond *syntactic modularity* that enables syntactical separation of code, our framework achieves *semantic modularity*, which allows the components to be modularly type-checked and separately compiled (see Section 2.1). Therefore, more errors could be detected statically and components can be distributed in binary.

1.1 Motivation

The world is complex and programmers encounter a variety of difficulties in software development. The tools they have are programming languages. However, most languages are only suitable in their own fields. For example, the C programming language is handy for systems development, but it would be painful to write web applications using its naked pointers.

Building more general languages does not solve the problem. Languages features are not always orthogonal, so that some advantages may need to be sacrificed to combine them. Moreover, the explosion of language features would make it hard to perform optimizations and reasoning on the language. Programmers would also have to be patient to learn and careful to use the language. Instead, we still need more specific languages to save the time of programmers and machines.

However, creating new languages requires both knowledge about the theory of programming language and the knowledge of the specific domain. Furthermore, heavy engineering effort is needed to build and maintain the language.

Language workbenches [20] are tools that facilitate implementation of (domain-specific) languages. They provide utilities to define, reuse and compose languages, together with auxiliary functionalities such as IDE support and debugging. Examples include Ensō [14], MetaEdit+ [37], MPS [43], Spoofax [36] and Xtext [17].

Language workbenches allow us to modularize languages as components. Thus, the effort of creating and maintaining languages could be reduced by sharing common parts among different languages. Nevertheless, for most of them, the representation and composition of languages achieve only a syntactical level of modularity. Semantical properties including modular type-checking and separate compilation of language components are not fully available. That makes the correctness heavily rely on the tool itself, and programmers do not have the flexibility to manipulate components directly. The need of global compilation is also unpleasant since it requires more time and textual level access to existing components.

We argue that, better techniques for representation and composition that ensure semantic modularity could make modularization of languages easier and safer. Gems, which uses modular external visitors [49] and type-safe modular parsing [70], is our attempt towards that goal.

1.2 Contributions

The contributions in the thesis are:

- **A pattern for modularizing syntax and semantics**: We propose a pattern for writing abstract syntax and semantics (operations on ASTs) in a modular way. It is based on modular external visitors [49], and we add several enhancements.
- **A technique for semantically modular parsing**[1]: We present a technique for building semantically modular parsers, which can evolve with the abstract syntax. We also identify challenges to achieve modularity in parsing.
- **A metaprogramming library for reducing boilerplate code**: We design and implement a metaprogramming library to automatically generate boilerplate code in modular language components, which is based on Scala's macro annotation [9] and Scalameta toolkit [55].

[1]As stated in the acknowledgements, it is a joint work [70] with Haoyuan Zhang.

- **A case study of TAPL interpreters**: We implement interpreters of the first 18 languages in *Types and Programming Languages* (TAPL) book [46]. They are compared with a non-modular implementation to investigate the extent of reuse and performance penalty of language modularization.

It is worth mentioning that, although we use Scala and metaprogramming in our implementation, the modularization techniques and patterns do not rely on metaprogramming and Scala-specific features such as case classes.

1.3 Outline

The thesis is structured as follows:

In Chapter 2, we introduces background knowledge, including the concept of modularity, the expression problem [66], the VISITOR pattern [23], Object Algebras [50], and parser combinators [8, 64].

In Chapter 3, we demonstrate an intuitive example of language modularization using Gems, in which we compose two small languages to create a new one.

In Chapter 4, we present modularization techniques of abstract syntax and semantics (operations on ASTs). We empoly modular external visitors [49], and add several enhancements.

In Chapter 5, we introduce semantically modular parsing, with traditional OO ASTs, Object Algebras, and visitors.

In Chapter 6, we combine all the techniques to create modular language components and illustrate code generation by metaprogramming.

In Chapter 7, we show a case study of interpreters in *Types and Programming Languages* (TAPL) book [46], and compare with a non-modular implementation.

In Chapter 8, we review related work and conclude.

Chapter 2

Background

This chapter introduces some concepts and techniques as background knowledge. Section 2.1 talks about syntactic and semantic modularity. Section 2.2 demonstrates the *expression problem* [66] as a challenge of pursuing modularity. Section 2.3 reviews the famous VISITOR design pattern in object-oriented programming. Section 2.4 introduces Object Algebras as a variant of VISITOR pattern and a lightweight solution of the expression problem. Section 2.5 discusses parser combinators and Scala's parser combinators library.

2.1 Modularity

The idea of dividing a whole software program into modular components was introduced decades ago [41]. Organizing code by various type of modules is encouraged and supported by many programming languages and frameworks. However, the term "modularity" is too general and too vague for discussion. Therefore, we explain the concept of modularity by dividing it into two levels: *syntactic* and *semantic* modularity.

Syntactic modularity refers to modular components on the syntactical level. Related modularization techniques allow programmers to write code separately. Several *Language Workbenches* [20, 36, 17] and many extensible parser generators [45, 25, 24] belong to this category. These approaches apply some sort of textual code composition and generation to glue code together. For instance, *superimposition* [1] combines so-called *feature structure trees* and generates corresponding code.

Although such approaches are relatively popular in practice, the correctness of code composition heavily depends on modularization tools, and errors are reported only after composition. Moreover, there is no explicit interface for modular reasoning about the program. These concerns lead to stronger models for modularity [35].

Semantic modularity requires the components to be modularly type-checked and separately compiled. It is on a higher level of modularity, and more related to the host programming languages. Type checking on individual components often reveals bugs earlier, before composing them together. Separate compilation reduces the cost of updating components, and also encourages independent development. Some examples for semantic modularity are *Family Polymorphism* [16] and solutions [50, 44] to the *expression problem* [66].

In this thesis, all modularization techniques we use are applied on the level of semantic modularity, and the word modularity implies semantic modularity in the following content unless explicitly specified.

2.2 The Expression Problem

The value of extensibility and modularity is evident, nevertheless it could be challenging to achieve them. As a classic and fundamental problem, the *expression problem* [66] illustrates the difficulty of preserving semantic modularity with extensions.

The problem appears when extending a recursive datatype and operations that consume it. There are two dimensions of extension. One is adding new data variants to the datatype. The other is adding new operations.

A solution must satisfy five requirements, which are listed below. The fifth one is added by Zenger and Odersky [44].

1. **Extensibility in both dimensions**: Adding both new operations and new data variants should be supported. Existing operations should be able to be updated to include new data variants.
2. **Strong static type safety**: Operations which do not cover all variants should not be applied on a datatype. Such errors should be spotted statically.
3. **No modification or duplication**: When extending from existing components, modification or duplication of old code is not allowed.
4. **Separate compilation and type-checking**: Only the new-added components need to be compiled and type-checked. Old ones are not reprocessed.
5. **Independent extensibility**: Non-linear extensions should be supported by allowing independent components to be created, added, or composed.

Using ordinary datatype representations, the two major programming paradigms Functional Programming (FP) and Object-Oriented Programming (OOP) each excel in one dimension of extension.

In functional programming, the recursive datatype is usually represented by Algebraic Data Type (ADT). ADT is also known as "sum of product types", as each alternative of the outermost sum type represents a data variant. Operations are naturally represented by functions. Therefore, it is easy to add new operations but hard to add new data variants.

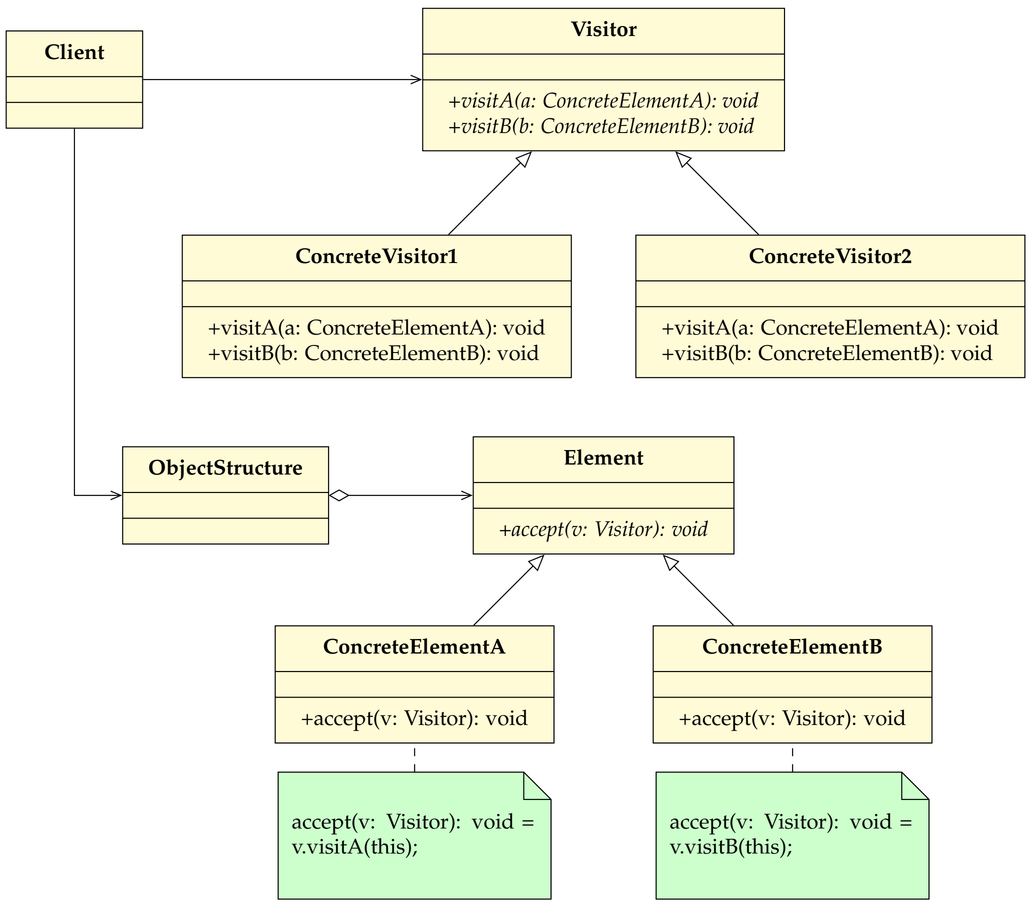

Client
Visitor
+visitA(a: ConcreteElementA): void
+visitB(b: ConcreteElementB): void
ConcreteVisitor1
+visitA(a: ConcreteElementA): void
+visitB(b: ConcreteElementB): void
ConcreteVisitor2
+visitA(a: ConcreteElementA): void
+visitB(b: ConcreteElementB): void
ObjectStructure
Element
+accept(v: Visitor): void
ConcreteElementA
+accept(v: Visitor): void
ConcreteElementB
+accept(v: Visitor): void
accept(v: Visitor): void = v.visitA(this);
accept(v: Visitor): void = v.visitB(this);

FIGURE 2.1: UML diagram of the VISITOR design pattern.

In object-oriented programming, the datatype is usually represented by an abstract class or interface, with abstract methods for the operations. Data variants are subclasses which implement those methods. Adding new data variants equals adding new subclasses, thus it is easy. However, adding new operations is hard since it requires adding methods in the whole class hierarchy.

2.3 The VISITOR Pattern

Instead of having operations entangled with class declarations, the VISITOR design pattern [23] allows us to write them separately. In terms of the expression problem, it swaps extensibility of the two dimensions for object-oriented languages. That is, with the VISITOR pattern, adding new operations is easy but adding new data variants is hard. Figure 2.1 shows this pattern as a UML diagram.

In the diagram, the *Visitor* interface is an abstraction of all *ConcreteVisitor*s. It has several *visit* methods, one for each *ConcreteElement*. Each *ConcreteVisitor* is a concrete operation with all the methods implemented.

```
trait Visitor[E] {
  def lit(x: Int): E
  def add(l: E, r: E): E
}
trait Exp {
  def apply[E](vis: Visitor[E]): E
}
case class Lit(x: Int) extends Exp {
  def apply[E](vis: Visitor[E]): E = vis.lit(x)
}
case class Add(l: Exp, r: Exp) extends Exp {
  def apply[E](vis: Visitor[E]): E =
    vis.add(l(vis), r(vis))
}
object Eval extends Visitor[Int] {
  def lit(x: Int): Int = x
  def add(l: Int, r: Int): Int = l + r
}
```

FIGURE 2.2: AST and evaluation using internal visitors.

Element is the abstract datatype, which has only an *accept* method taking a *Visitor*. Each *ConcreteElement* represents a data variant, with its special *accept* method to select the corresponding *visit* method to call when being visited.

We can simply add a new operation as a *ConcreteVisitor* by extending the *Visitor* interface and implementing the methods for every data variant. It will not break modularity. However, adding new data variants becomes a problem, since it requires modifying the *Visitor* interface to include a new *visit* method.

Internal and External Visitors If the datatype is recursive, operations often need recursive calls on subtrees. Regarding where to place recursive calls, visitors can be divided into *internal* and *external* visitors [7].

We will illustrate their difference by writing the *abstract syntax tree* (AST) and an evaluation operation for an expression language consisting of literals and additions. The grammar is below.

```
<exp> ::= literal
        | <exp> "+" <exp>
```

Internal visitors do the recursive calls in *accept* methods by feeding the visitor argument to subtrees, so that concrete visitors (operations) only combine recursive results. Figure 2.2 shows AST and evaluation operation written by internal visiters. For more concise code, we use Scala's special method name `apply` instead of "accept", thus arguments can be passed without an explicit method call.

```
trait Visitor[E] {
  def lit(x: Int): E
  def add(l: Exp , r: Exp ): E
}
trait Exp {
  def apply[E](vis: Visitor[E]): E
}
case class Lit(x: Int) extends Exp {
  def apply[E](vis: Visitor[E]): E = vis.lit(x)
}
case class Add(l: Exp, r: Exp) extends Exp {
  def apply[E](vis: Visitor[E]): E = vis.add( l , r )
}
object Eval extends Visitor[Int] {
  def lit(x: Int): Int = x
  def add(l: Exp , r: Exp ): Int = l(Eval) + r(Eval)
}
```

FIGURE 2.3: AST and evaluation using external visitors.

External visitors allow concrete visitors to control the recursive calls by themselves. Instead of taking recursive results, they take subtrees at recursive positions. Therefore, it is more flexible to choose when and wether to perform recursive calls, or even delegate to other visitors. Figure 2.3 shows the version using external visitors, with difference highlighted as grey.

Comparing the two variants, internal visitors are more simple when writing concrete operations, whereas external visitors are more flexible and powerful in terms of performing recursive traversal and expressing dependencies. Because of that, external visitors are more suitable for language ASTs and related semantic operations.

For example, *if-then-else* expressions are very common in programming languages. Considering evaluation, if we use internal visitors, both branches will be evaluated and one result is discarded finally. By contrast, with external visitors we can decide which branch to evaluate according to the condition value. It is not only more efficient but also obeys the semantics. Furthermore, evaluation may cause side effects in some cases so that unnecessary recursive calls must be eliminated.

2.4 Object Algebras

Object Algebras [50] were proposed as a solution to the expression problem. It is a lightweight design pattern and only requires simple generics and (multiple) inheritance, which are ordinary features in object-oriented programming. Therefore, it can be easily encoded in mainstream OO languages such as Java.

Object Algebras are closely related to Church encodings and internal visitors. We will continue using the example in the last section, and compare with the code of internal visitors in the following demonstration.

Object Algebra Interface An *Object Algebra interface* is just an internal visitor interface, which captures all variants of the datatype as abstract methods.

```
trait Alg[E] {
  def lit(x: Int): E
  def add(l: E, r: E): E
}
```

The code above is as same as the visitor interface shown in Figure 2.2, with only the name changed.

Object Algebra (Operation) A concrete *Object Algebra* represents an operation by implementing the Object Algebra interface, which is similar to a concrete visitor. The only difference here is we are using `trait` instead of `object` for extensibility.

```
trait Eval extends Alg[Int] {
  def lit(x: Int): Int = x
  def add(l: Int, r: Int): Int = l + r
}
```

This type of extension does not introduce new methods, but only implements all abstract methods declared in the Object Algebra interface. In the example above, methods `lit` and `add` are inherited from `Alg` and implemented in `Eval`.

Adding New Variants Now let us consider adding subtraction expressions to the language. The new grammar is shown below.

```
<exp> ::= literal
        | <exp> "+" <exp>
        | <exp> "-" <exp>
```

A new data variant must be added to incorporate the new syntax structure. Thus, a new Object Algebra interface `AlgSub` is created by extending from `Alg`, with a new method `sub` added.

```
trait AlgSub[E] extends Alg[E] {
  def sub(l: E, r: E): E
}
```

Different from the previous extension of `Eval`, this type of extension introduces new abstract methods without concrete definitions.

After that, the old evaluation operation `Eval` must be patched for subtraction expressions. This can be achieved by create a new Object Algebra called `EvalSub`, which implements the interface `AlgSub` and inherits from `Eval` for code reuse.

```
trait EvalSub extends AlgSub[Int] with Eval {
  def sub(l: Int, r: Int): Int = l - r
}
```

Adding New Operations Adding new operations is easy since it equals creating new Object Algebras. For instance, we write a printing operation `PrintSub` to print ASTs.

```
trait PrintSub extends AlgSub[String] {
  def lit(x: Int): String = x.toString
  def add(l: String, r: String): String = s"($l + $r)"
  def sub(l: String, r: String): String = s"($l - $r)"
}
```

Instances of the Datatype In contrast with internal visitors, Object Algebras do not have a concrete representation for the datatype. Instances are created by generic methods which take a concrete algebra. The following code shows representations of two expressions.

```
// 2 + 3
def exp1[E](alg: Alg[E]): E = alg.add(alg.lit(2), alg.lit(3))
// 1 - (2 + 3)
def exp2[E](alg: AlgSub[E]): E = alg.sub(alg.lit(1), exp1(alg))
```

Operations are applied by feeding Object Algebras to those generic functions. We can evaluate `exp1` and print `exp2` as below.

```
val i: Int = exp1(new Eval {})
val s: String = exp2(new PrintSub {})
```

Discussion As a lightweight approach, Object Algebras are quite practical in OO languages such as Java. There have been several following research works towards applications [47, 24, 4] and better usage [52, 71] of Object Algebras. Especially, there were some attempts to modularize language syntax and semantics [24, 30]. However, some deficiencies have been pointed out in those works as well. We summarize into two points.

Firstly, similarly to internal visitors, Object Algebras have a fixed pattern of recursion which makes it hard to express dependencies. Thus, some operations can be written in a very elegant way, while some others are awkward to implement.

Secondly, there is no concrete representation of the datatype. It is very hard to store and pass the instances in traditional OO languages like Java, since they are written as generic methods. This may also lead to efficiency problems. In the work of NOA [24], they found that the straightforward implementation of parsers would require re-parsing the input for applying every operation, and their partial solution relies on runtime type information (Java's `instanceof`).

However, we must point out, it is not a technical issue, but a limitation of current languages like Java and Scala. Specifically, they do not have full integration of function subtyping and polymorphism. Thinking in a functional way, the type of previous `exp1` (and all the other expressions of the `Alg` algebra) is just a polymorphic function type `forall E. Alg[E] => E`. In an ideal language, we could have such standalone functions as concrete datatypes, and subtyping relationship could be straightforwardly derived.

2.5 Parser Combinators

Parsing is fundamental to computer programming and it has been heavily studied. There are two major techniques of building parsers: *parser generators* and *parser combinators* [8, 64]. A parser generator tool often provides a DSL for specifying syntax, and then parsers are generated automatically. A parser combinator library usually does not have explicit syntax specifications. Instead, small parsers are glued together by *combinators* in some host languages.

Parser combinators can be traced to the work of Burge [8] in 1975, and over the years many works have been conducted [64, 28, 18, 29]. Parser combinators are very popular in functional programming, where parsers are represented by functions, and combinators are represented by higher-order functions. One famous parser combinator library is Parsec [39] in Haskell. Because all parsers and combinators are first class values in the host language, programmers have great flexibility to manipulate and customize parsing. It also avoids integration of different tools and languages as parser generators often required by [27].

Scala has a standard parser combinator library [54]. Table 2.1 shows some common parser combinators we selected from the library documentation. The alternative combinator `|`, the sequential concatenation combinator `~`, and the application combinator `^^` are especially frequently used among them.

Figure 2.4 shows an example of parsing addition expressions using the library. The object `Parser` is extended from `StandardTokenParsers` to inherit basic parsers and combinators. In line 4 we define delimiters for lexical analysis. From line 6 to 12, we write three parsers `pLit`, `pAdds`, and `pExp`. The types in the bracket of `Parser[]` are types of parsing results. In line 12, the result of `pLit` and `pAdds` are extracted and summed up.

def `*: Parser[List[T]]`	Returns a parser that repeatedly parses what this parser parses.
def `?: Parser[Option[T]]`	Returns a parser that optionally parses what this parser parses.
def `+: Parser[List[T]]`	Returns a parser that repeatedly (at least once) parses what this parser parses.
def `~[U](q: => Parser[U]): Parser[~[T, U]]`	A parser combinator for sequential composition.
def `^^[U](f: (T) => U): Parser[U]`	A parser combinator for function application.
def `^^^[U](v: => U): Parser[U]`	A parser combinator that changes a successful result into the specified value.
def `<~[U](q: => Parser[U]): Parser[T]`	A parser combinator for sequential composition which keeps only the left result.
def `~>[U](q: => Parser[U]): Parser[U]`	A parser combinator for sequential composition which keeps only the right result.
def `|[U >: T](q: => Parser[U]): Parser[U]`	A parser combinator for alternative composition.
def `|||[U >: T](q0: => Parser[U]): Parser[U]`	A parser combinator for alternative with longest match composition.

TABLE 2.1: Common combinators in Scala's parser combinator library [54].

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

object Parser extends StandardTokenParsers {
  lexical.delimiters += ("(", ")", "+")

  lazy val pLit: Parser[Int] =
    numericLit ^^ { _.toInt }
  lazy val pAdds: Parser[List[Int]] =
    ("+" ~> pExp).*
  lazy val pExp: Parser[Int] =
    "(" ~> pExp <~ ")" |
    pLit ~ pAdds ^^ { case a ~ as => as.foldLeft(a)((x, y) => x + y) }
  def parse(input: String): Int =
    phrase(pExp)(new lexical.Scanner(input)).get
}
```

FIGURE 2.4: An example of using parser combinators in Scala.

Finally, a method `parse` is defined as a wrapper to parse input strings. In client code, the parser can be used as below.

```
val x: Int = Parser.parse("1 + (2 + 3)")   // 6
```

Here we directly calculate an integer as the parsing result, while normally the result of parsing is an AST of the language. It brings challenges of reusing parsers when the language evolves. In Chapter 5, we will discuss more about parsing in terms of modularity and extensibility.

Chapter 3

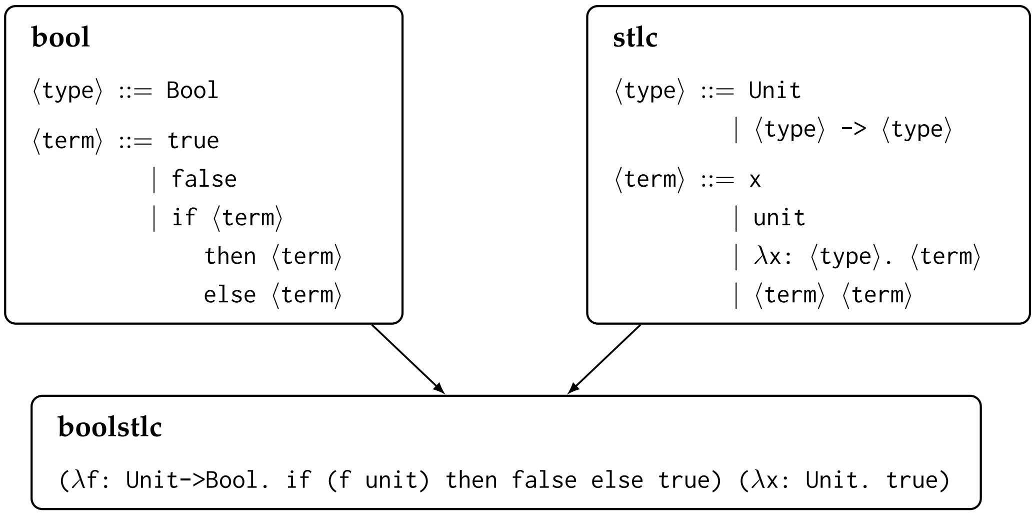
Collecting Gems

In this chapter we will demonstrate an example of language modularization using Gems, as an overview and intuition to technical details in the following chapters. The example is to compose two self-contained languages:

- `bool`
  The language includes boolean literals and if-then-else expressions, with a simple boolean type. Some example expressions in the language are:
  ```
  if true then false else true
  if false then (if true then false else true) else true
  ```

- `stlc`
  The language includes basic structures of the simply type lambda calculus, with the unit value and type. Some example expressions in the language are:
  ```
  (λx: Unit. x) unit
  (λf: Unit->Unit. f unit) (λx: Unit. x)
  ```

The two languages are written using Gems and composed together to build a new language `boolstlc`. It combines the power of both `bool` and `stlc`, and thus allows us to write more complex expressions. The following figure illustrates the composition.



bool

```
⟨type⟩ ::= Bool
⟨term⟩ ::= true
         | false
         | if ⟨term⟩
             then ⟨term⟩
             else ⟨term⟩
```

stlc

```
⟨type⟩ ::= Unit
         | ⟨type⟩ -> ⟨type⟩
⟨term⟩ ::= x
         | unit
         | λx: ⟨type⟩. ⟨term⟩
         | ⟨term⟩ ⟨term⟩
```

boolstlc

```
(λf: Unit->Bool. if (f unit) then false else true) (λx: Unit. true)
```

As we said, the two languages `bool` and `stlc` are self-contained. Except the abstract syntax, they may have their own parsers, typers, evaluators, etc. In this example, we are not only composing their syntax, but their whole interpreter pipelines. Thus, the result of composition is a comprehensive implementation of the new language.

For the sake of brevity, we list the full code online [22], and only show important code in the demonstration. Readers may focus on the overall structure and skip details of the code in the first reading, since they will be discussed in later chapters.

Basically we will demonstrate two aspects of language modularization using Gems: how to write a language in a modular way so that they can be reused, and how to compose those languages to create a new one. In Section 3.1, we implement the abstract syntax, operations (semantics), and parser of language `bool` using Gems. In Section 3.2 we first present the abstract syntax of language `stlc`. Other parts of `stlc` are omitted for brevity since they are similar with `bool`. Then we demonstrate the composition to build `boolstlc`, which can be done with a small amount of code. In Section 3.3 we show an example of building an imaginary DSL from a base language with direct extensions.

3.1 Implementation of the Language `bool`

We take the language `bool` as an example to demonstrate how to write a modular language that is open for future reuse.

Abstract Syntax The abstract syntax of the language `bool` is implemented in Figure 3.1. We have two visitor interfaces `Term` and `Type` containing the corresponding variants of the abstract syntax. At recursive positions of the datatype, the type parameter `R` is used instead of a concrete type for extensibility.

With the interfaces defined, ASTs of language `bool` are captured by type `Exp[Term]` and `Exp[Type]`. The technique we use is based on modular external visitors [49]. Details about the technique will be discussed in Section 4.2, including the `apply` method for managing recursive calls.

The two macro annotations of `Lang` are required by Gems for generating companion objects of the two interfaces, that contain utilities which can reduce boilerplate code. The name of the language is passed as a string (in this case `"bool"`), which should correspond with the package name. Table 3.1 shows structures in the generated companion object of `Term`. Code generation will be discussed in Section 6.3.

Operations Figure 3.2 shows the implementation of a small-step evaluation operation. There are two concrete visitors `Eval` and `IsVal`. `Eval` is mixed with a interface `IIsVal` which has the following definition:

Name	Generated structures

Factory classes	`case class TmTrue[A[-X, Y] <: Term[X, Y], B[-R, _]]()`
	`case class TmFalse[A[-X, Y] <: Term[X, Y], B[-R, _]]()`
	`case class TmIf[A[-X, Y] <: Term[X, Y], B[-R, _]](..)`

Factory wrapper	`object Factory`

Traversal templates	`trait Query[-R, E]`
	`trait Transform[A[-X, Y] <: Term[X, Y]]`

Lifter	`trait Lifter[-R, E, C]`
Type conversion	`trait Convert[A[-X, Y] <: Term[X, Y]]`

TABLE 3.1: Generated structures for Term in Figure 3.1.

```
trait IIsVal[A[-R, E]] {
  def isVal: A[Exp[A], Boolean]
}
```

The field indicates that the evaluation depends on the IsVal operation to decide whether a term is a value and thus cannot be further reduced. Section 4.3 discusses more about operation dependencies.

Eval is also mixed with trait Term.Convert, which is generated by Gems for type conversion. It provides a method convertBool which is used in line 12. After type conversion, we are able to analyse the condition term of if-then-else, namely e1, by applying an anonymous visitor from line 13 to 18. Such case analysis is useful in practice since we often need to perform ad-hoc operations and we certainly do not want to write them as named, top-level operations. Section 4.7 talks about this issue. The apply method is used for performing recursive calls in line 21.

The IsVal operation extends from a generated traversal template Term.Query. The template sets all cases to a default value beforehand, and thus we only need to override interesting cases instead of repeating the fallback value everywhere.

Parser Parser of language bool is implemented in Figure 3.3. It uses Packrat parsing [19] utilities in Scala's parser combinator library. It has a quite similar structure to the example in Section 2.5. There are two fields pBoolE and pBoolT for parsing terms and types, and two aliases pE and pT for them. When perform recursively parsing, those aliases are used. That is the key for extensibility. Details about semantically modular parsers will be discussed in Chapter 5.

3.2 Composition for the Language boolstlc

Figure 3.4 shows the abstract syntax of language stlc. It also has two sorts: terms and types. However, it is different from bool because the two sorts are nested. Therefore in the interface Term we define an extra type parameter T for type occurrences.

```
package language.bool
import gems._

@Lang("bool")
trait Term[-R, E] {
  def tmTrue(): E
  def tmFalse(): E
  def tmIf(e1: R, e2: R, e3: R): E
  def apply(e: R): E
}

@Lang("bool")
trait Type[-R, E] {
  def tyBool(): E
  def apply(e: R): E
}
```

FIGURE 3.1: The abstract syntax of language bool using Gems.

```
package language
package bool
import gems._
import Term.Factory._

trait Eval[A[-X, Y] <: Term[X, Y]] extends Term[Exp[A], Exp[A]]
  with IIsVal[A] with Term.Convert[A] {
  def tmTrue(): Exp[A] = TmTrue[A, A]()
  def tmFalse(): Exp[A] = TmFalse[A, A]()
  def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
    if (e1(isVal)) {
      val c = convertBool(e1).getOrElse(cnvFailed)
      c(new Term[Exp[A], Exp[A]] {
        def tmTrue(): Exp[A] = e2
        def tmFalse(): Exp[A] = e3
        def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] = runtimeError
        def apply(e: Exp[A]): Exp[A] = impossible
      })
    }
    else
      TmIf[A, A](apply(e1), e2, e3)
}

trait IsVal[A[-R, _]] extends Term.Query[Exp[A], Boolean] {
  override def default: Boolean = false
  override def tmTrue(): Boolean = true
  override def tmFalse(): Boolean = true
}
```

FIGURE 3.2: The evaluator of language bool using Gems.

```
package language
package bool
import gems._
import Term.Factory._
import Type.Factory._

trait Parse[A[-X, Y] <: Term[X, Y], B[-X, Y] <: Type[X, Y]] extends BaseParser {
  lexical.reserved += ("true", "false", "if", "then", "else", "Bool")
  lexical.delimiters += ("(", ")")
  private lazy val pTrue = "true" ^^^ TmTrue[A, A]()
  private lazy val pFalse = "false" ^^^ TmFalse[A, A]()
  private lazy val pIf =
    ("if" ~> pE) ~ ("then" ~> pE) ~ ("else" ~> pE) ^^ { case e1 ~ e2 ~ e3 =>
    TmIf[A, A](e1, e2, e3) }
  lazy val pBoolE: PackratParser[Exp[A]] =
    pTrue ||| pFalse ||| pIf ||| "(" ~> pE <~ ")"
  lazy val pBoolT: PackratParser[Exp[B]] =
    "Bool" ^^^ TyBool[B, B]()
  lazy val pE: PackratParser[Exp[A]] = pBoolE
  lazy val pT: PackratParser[Exp[B]] = pBoolT
}
```

FIGURE 3.3: The parser of language bool using Gems.

Such multi-sorted syntax is discussed in Section 4.5.

The other parts of language stlc are omitted because they have similar structure to bool. Full code is listed online [22].

It is easy to merge the two languages bool and stlc using our framework, and the code is neat because the features from the two languages do not interfere with each other. We still demonstrate the composition from three aspects: the abstract syntax, operations, and parsers.

Figure 3.5 illustrates composition to build the abstract syntax of boolstlc. It only requires 4 lines of code excluding the imports. The visitor interfaces are composed correspondingly for each sort using multiple inheritance.

Figure 3.6 shows composition of operations. For each operation, it takes 2 to 3 lines to extend from the parent languages with auxiliary utilities. If we want to redefine some cases we could override them in the new trait, but here we do not need to do so.

Figure 3.7 composes parsers. Comparing with other parts, it needs a bit more code to select the corresponding fields in the two parent parsers. However it is still straightforward and simple.

As we can see, if we have modularized language as components, it is easy to quickly build a new language by composing the related elements. We also have great flexibility to modify the language since every part can be customized as needed in composition. Furthermore, the compositions are statically type-safe in the host language

```
package language.stlc
import gems._

@Lang("stlc")
trait Term[-R, E, -T] {
  def tmVar(x: String): E
  def tmApp(e1: R, e2: R): E
  def tmAbs(x: String, t: T, e: R): E
  def tmUnit(): E
  def apply(e: R): E
}

@Lang("stlc")
trait Type[-R, E] {
  def tyArr(t1: R, t2: R): E
  def tyUnit(): E
  def apply(t: R): E
}
```

FIGURE 3.4: The abstract syntax of language `stlc` using Gems.

```
package language
package boolstlc
import gems._

@Lang("boolstlc")
trait Term[-R, E, -T] extends bool.Term[R, E] with stlc.Term[R, E, T]

@Lang("boolstlc")
trait Type[-R, E] extends bool.Type[R, E] with stlc.Type[R, E]
```

FIGURE 3.5: Composition of abstract syntax for language `boolstlc`.

and components can be separately compiled.

3.3 Direct Extensions and New DSLs

The previous example shows how to define two independent language components and merge them to easily form a single language. It belongs to the category of language unification [15]. Another kind of composition, which is perhaps more common, is to enrich a base language with some (domain-specific) extensions directly. The base language could be explicit, which means it has been built with a unique name, or implicit, which means it is in-place merged from components.

We show an imaginary toy language that aims at making shell scripting easier. The language is extended from an implicit base language, which consists of three components: `boolstlc` for functions and booleans, `record` for records, and `string` for strings. The last two have similar corresponding ones in the case study. Thus the language

```
package language
package boolstlc
import gems._

trait Eval[A[-R, E, -F] <: Term[R, E, F], T]
  extends Term[Exp2[A, T], Exp2[A, T], T]
  with bool.Eval[A[-?, ?, T]] with stlc.Eval[A, T] with Term.AllChains[A, T]

trait IsVal[A[-R, E, -F], T] extends Term.Query[Exp2[A, T], Boolean, T]
  with bool.IsVal[A[-?, ?, T]] with stlc.IsVal[A, T]

trait Subst[A[-R, E, -F] <: Term[R, E, F], T] extends Term.Transform[A, T]
  with stlc.Subst[A, T]

trait Print[A[-R, E, -F], T] extends Term[Exp2[A, T], String, T]
  with bool.Print[A[-?, ?, T]] with stlc.Print[A, T]

trait PrintT[A[-R, _]] extends Type[Exp[A], String]
  with bool.PrintT[A] with stlc.PrintT[A]
```

FIGURE 3.6: Composition of operations for language boolstlc.

```
package language
package boolstlc
import gems._

trait Parse[A[-R, E, -F] <: Term[R, E, F], B[-X, Y] <: Type[X, Y]]
  extends stlc.Parse[A, B] with bool.Parse[A[-?, ?, Exp[B]], B] {
  lazy val pBoolstlcE: PackratParser[Exp2[A, Exp[B]]] = pBoolE ||| pStlcE
  lazy val pBoolstlcT: PackratParser[Exp[B]] = pBoolT ||| pStlcT
  override lazy val pE: PackratParser[Exp2[A, Exp[B]]] = pBoolstlcE
  override lazy val pT: PackratParser[Exp[B]] = pBoolstlcT
}
```

FIGURE 3.7: Composition of parsers for language boolstlc.

has first class functions and a simple type system. Some other structures are added to the language to execute shell commands and write log.

The code below shows an example of writing a backup script using the language.

```
fn backup(from: String, to: String) = {
  log (exec "date").result
  if (exec "cp -r $from $to").success
    then log "Successfully backed up: $from"
    else log "Failed to back up: $from"
}
backup from "~/docs/" to "~/backup/"
```

The language has a simple syntactic sugar, the dollar sign `$`, for embedding variables into a string. Two built-in functions `exec` and `log` allow the user to execute a shell command and write log. `exec` returns a record of boolean and string, representing the exit status and result. Function applications have a special syntax that parameter names can be put before the arguments, as indicated by the last line. Other parts are straightforward.

The abstract syntax of the language would be built as the code below, in which we only need to introduce the built-in functions as the domain-specific part.

```
@Lang("mydsl")
trait Term[-R, E, -T] extends boolstlc.Term[R, E, T] with record.Term[R, E]
  with string.Term[R, E] {
  def TmExec(e: R): E
  def TmLog(e: R): E
}
@Lang("mydsl")
trait Type[-R, E] extends boolstlc.Type[R, E] with record.Type[R, E]
  with string.Type[R, E]
```

Semantics of the components could be mostly reused in this language, but the parsers need to be rewritten since the concrete syntax is changed.

This imaginary example shows that, if we had a rich library of language components, it would be convenient to build a DSL by importing components from the library and adding domain-specific structures. We already have some finished components in our case study, listed in Table 7.2. Although they are probably not robust enough to build real languages, they provide a starting point to explore the idea.

Chapter 4

Modular Syntax and Semantics

This chapter presents how we modularize ASTs and semantic operations in Gems. We adopt modular external visitors [49] and make several enhancements. Section 4.1 introduces the *expression families problem* [49]. Section 4.2 shows the original work of modular external visitors. The subsequent sections demonstrate our enhancements. In Section 4.3 and 4.4 we borrow the idea from EVF [72] to deal with operation dependencies and have flexible recursive calls. In Section 4.5 we show how to extend abstract syntax with multiple sorts. In Section 4.6 we present context propagation for further extensibility, which is inspired by *implicit context propagation* [30]. In Section 4.7 we discuss how to perform case analysis on ASTs.

4.1 Expression Families Problem

Based on the expression problem (Section 2.2), *expression families problem* [49] is a stronger version that has more requirements. While satisfying the requirements of the expression problem, one solution must also have the following properties.

1. **Distinguishable types**: A combination of data variants should have its own distinguishable type. Thus, new datatypes extended from old ones have different identities in the type system.

2. **Subtyping of components**: Those different datatypes, together with their related operations, should have proper subtyping relationships.

3. **Full composability**: With all the data variants and operations, all combination of them should be allowed and flexibly composed.

For example, if we represent datatypes by sum types, when `Exp2` adds more variants on `Exp1`, `Exp2` is a supertype of `Exp1`. Thus, for any type `T`, `Exp2 => T` is a subtype of `Exp1 => T`, which indicates that operations on `Exp2` are able to consume `Exp1` as well.

Therefore, we have a family of related datatypes during extensions. The distinguishable types enable us to specify the "version" as typing constraints. The subtyping relationship guarantees that we cannot apply operations unsafely. In terms of language modularization, those properties are desired.

4.2 Modular External Visitors

As a solution to the expression families problem, modular external visitors [49] exploit *type variance* to maintain the subtyping relationship during extension. Type variance is well supported in Scala, represented by annotations before type parameters. The plus and minus (+ and -) annotations mean *covariance* and *contravariance*, respectively. We will introduce modular external visitors by demonstrating differences from the traditional external visitors.

4.2.1 Visitor Interface

Similarly to traditional external visitors, a visitor interface with method signatures is used to abstract over concrete visitors. However, names of the concrete datatype at recursive positions are replaced by a contravariant type parameter. In the example below, it is `-R`.

```
trait IAdd[-R, E] {
  def lit(x: Int): E
  def add(l: R, r: R): E
}
```

Comparing with the code in Figure 2.3, this change decouples the visitor interface with the concrete datatype. Thus, it is easy and straightforward to extend the visitor interface for adding new variants.

```
trait ISub[-R, E] extends IAdd[R, E] {
  def sub(l: R, r: R): E
}
```

4.2.2 Datatype

In the VISITOR pattern, the datatype has an `accept` (we use `apply` in our examples) method which takes a concrete visitor and returns the result value. The code below shows two datatypes `AddExp` and `SubExp`, corresponding to the two visitor interfaces `IAdd` and `ISub`.

```
trait AddExp {
  def apply[E](vis: IAdd[AddExp, E]): E
}
trait SubExp {
  def apply[E](vis: ISub[SubExp, E]): E
}
```

The problem is, although the two interfaces have subtyping relationship, the two datatypes are not related at all. Writing different datatypes for each interface is not acceptable, since it does not meet the requirements of the expression families problem. Instead, we abstract the visitor interfaces as a type parameter, so that they can share a common datatype.

```
trait Exp[-A[-R, _]] {
  def apply[E](vis: A[Exp[A], E]): E
}
```

The type parameter `-A` above represents any visitor interface, and `Exp[A]` is the corresponding datatype. `Exp` can be regarded as a fixed-point operator on the type level, that builds the recursive type `Exp[A]` for interface `A`.

The contravariance of `A` guarantees the subtyping relationship is preserved. For instance, we rewrite `AddExp` and `SubExp` as type synonyms as below. `AddExp` is now a subtype of `SubExp`, since `IAdd` is a supertype of `ISub`.

```
type AddExp = Exp[IAdd]
type SubExp = Exp[ISub]
```

4.2.3 Factories

Factories of traditional external visitors contain hardcoded datatype names, thus they are not reusable. Similar to the abstraction of the datatype, the visitor interfaces can be passed into factories as a type parameter. It is constrained by an upper type bound to express that the interface must include certain cases.

In the example below, two factories `Lit` and `Add` have a type parameter `A` which must be a subtype of `IAdd`, thus the visitor interface represented by `A` contains `lit` and `add` methods.

```
case class Lit[A[-X, Y] <: IAdd[X, Y]](x: Int) extends Exp[A] {
  def apply[E](vis: A[Exp[A], E]): E = vis.lit(x)
}
case class Add[A[-X, Y] <: IAdd[X, Y]](e1: Exp[A], e2: Exp[A]) extends Exp[A] {
  def apply[E](vis: A[Exp[A], E]): E = vis.add(e1, e2)
}
```

A factory class `Sub` for the subtraction case can be defined similarly, with only the corresponding names changed.

```
case class Sub[A[-X, Y] <: ISub[X, Y]](e1: Exp[A], e2: Exp[A]) extends Exp[A] {
  def apply[E](vis: A[Exp[A], E]): E = vis.sub(e1, e2)
}
```

With the factories, it is easy to construct concrete expressions. Notice the subtyping relationship allows us to reuse `e1` in `e2` straightforwardly.

```
val e1: AddExp = Add(Lit(1), Lit(2))    // 1 + 2
val e2: SubExp = Sub(e1, Lit(3))        // (1 + 2) - 3
```

4.2.4 Concrete Visitor

Concrete visitors or operations are implementations of the visitor interface. For extensibility, concrete visitors should not have concrete datatypes hardcoded in them. The example below shows an evaluation visitor which implements the `IAdd` interface. Notice the datatype is abstract.

```
trait EvalAdd[A[-R, _]] extends IAdd[Exp[A], Int] {
  self: A[Exp[A], Int] =>
  def lit(x: Int): Int = x
  def add(l: Exp[A], r: Exp[A]): Int = l(this) + r(this)
}
```

The *self type* feature of Scala is used here to explicitly denotes the fact that, in the future, the whole object has this trait mixed-in has the type `A[Exp[A], Int]`. It is necessary because the object itself (represented by *self reference*, namely `this`) is passed to subtrees which have type `Exp[A]` for every recursive call.

To apply an operation, we first instantiate it as an object, then feed it to an expression. As an example, here we evaluate the expression `e1` defined in Section 4.2.3.

```
val e1: AddExp = Add(Lit(1), Lit(2))    // 1 + 2
object EvalAddObj extends EvalAdd[IAdd]
val v1: Int = e1(EvalAddObj)            // 3
```

New visitors could be built by extending from old ones, thus code are reused.

```
trait EvalSub[A[-R, _]] extends ISub[Exp[A], Int] with EvalAdd[A] {
  self: A[Exp[A], Int] =>
  def sub(l: Exp[A], r: Exp[A]): Int = l(this) - r(this)
}
```

4.3 Operation Dependencies

It is common in practice that one operation depends on others. Evaluation operation probably depends on substitution, and pretty-printing operation may depend on precedence of syntax structures. The use of external visitor makes it easy to apply other operations, because we have direct access to subtrees of AST.

To express such dependencies, we could use the similar style of EVF [72], that having abstract declarations in visitors. Each declaration represents a required operation that can be used in the current one.

For instance, when printing ASTs we often need to add parentheses. We do not want to add parentheses everywhere since it would pollute the result. One simple strategy is that we only add parentheses if a subtree has lower or equal precedence than the current. Precedence is defined for every syntax structure, and it is another separate operation that can be used somewhere else.

The code below presents a printing operation for the `IAdd` interface shown before. It depends on a precedence operation to add parentheses.

```
 1 trait Print[A[-X, Y] <: IAdd[X, Y]] extends IAdd[Exp[A], String] {
 2   self: A[Exp[A], String] =>
 3   val prec: A[Exp[A], Int]          // Dependency declaration
 4   private def addParens(f: Boolean, s: String): String =
 5     if (f) s"($s)" else s
 6   def lit(x: Int): String = x.toString
 7   def add(l: Exp[A], r: Exp[A]): String = {
 8     val myPrec = prec.add(l, r)
 9     val pl = addParens(l(prec) <= myPrec, l(this))
10     val pr = addParens(r(prec) <= myPrec, r(this))
11     s"$pl + $pr"
12   }
13 }
```

In line 3, the `prec` field for precedence represents the dependency. It remains abstract until `Print` is instantiated. From line 7 to line 12, the `add` method uses `prec` to obtain precedence of subtrees and compare with the precedence of "add". Then it can decide whether to add parentheses or not. The constraint of type parameter `A` limits it to be a subtype of `IAdd`. That guarantees the `add` method exists in `prec`.

An operation `Precedence` for syntax precedence is defined as follow, together with an object `Print` to instantiate the printing operation.

```
trait Precedence[A[-R, _]] extends IAdd[Exp[A], Int] {
  def lit(x: Int): Int = 10
  def add(l: Exp[A], r: Exp[A]): Int = 1
}
object Print extends Print[IAdd] {
  val prec: IAdd[Exp[IAdd], Int] = new Precedence[IAdd] {}
}
```

Literals will not be surrounded by parentheses because of the higher precedence.

```
val e: Exp[IAdd] = Add(Lit(1), Add(Lit(2), Lit(3)))
val s: String = e(Print)   // 1 + (2 + 3)
```

When abstract syntax evolves, operations and dependencies can be extended with semantic modularity. Dependencies are inherited in extensions. Moreover, we can change the implementations of dependencies since they are hidden behind the field declarations.

A better approach to declare dependencies is creating a distinct interface for each operation. Thus, other operations only need to extend the interface. It ensures consistency of names in multiple inheritance, and dependencies are more explicitly expressed. The following code illustrates this pattern.

```
trait DepOfPrec[A[-R, _]] {
  val prec: A[Exp[A], Int]
}
trait Print[A[-X, Y] <: IAdd[X, Y]] extends IAdd[Exp[A], String]
  with DepOfPrec[A] {
    ...
}
```

4.4 Abstracting Recursive Calls

In the previous sections, we showed how to use *self type* and *self reference* for making recursive calls in concrete visitors. However, an alternative approach using abstract and indirect recursive calls, as presented in EVF [72], is possibly better under certain circumstances.

Instead of feeding the self reference `this` to subtrees, this approach abstracts them as calls of a specific method. For example, we rewrite the `EvalAdd` in Section 4.2.4 as below, making recursive calls by calling a method `apply`.

```
trait EvalAdd[A[-R, _]] extends IAdd[Exp[A], Int] {
  def lit(x: Int): Int = x
  def add(l: Exp[A], r: Exp[A]): Int = apply(l) + apply(r)
}
```

The `apply` method is declared in abstract visitor interface `IAdd`. It accepts a subtree and returns the result of applying the recursive call on that subtree.

```
trait IAdd[-R, E] {
  def lit(x: Int): E
  def add(l: R, r: R): E
  def apply(x: R): E
}
```

The implementation of `apply` remains undefined until concrete visitor instances are created. When creating an instance, `apply` is implemented, often by simply feeding the self reference `this` like below.

```
object EvalAdd extends EvalAdd[IAdd] {
  def apply(x: Exp[IAdd]): Int = x(this)
}
```

There are two advantages of using this approach.

Firstly, and the most important, the abstraction of recursive calls gives us the chance to delay and redefine its meaning. For example, later in Section 4.6 we delegate the recursive calls of the inner visitor to the outer for context propagation. Without such abstraction, it would be impossible to achieve that.

Secondly, the special name `apply` we use, enables both syntax `datatype(operation)` and `operation(datatype)` when performing an operation on a datatype. Sometimes one is more natural than the other. For example, we could have:

```
val b: Bool = expr(IsValue)
val v: Value = Evaluate(expr)
```

Furthermore, this approach eliminates the annoying self type declaration at the beginning of every concrete visitor and it is lightweight in terms of coding.

4.5 Multi-Sorted Syntax

In the previous examples, we only have one extensible datatype representing the abstract syntax tree. In other words, the syntax has only one *sort*: expressions. However, multiple sorts are often needed in practical languages, such as expressions, types, kinds and so on. The original work of modular external visitors [49] did not discuss this issue.

As an instance, the syntax below shows a language which has two sorts: expressions and types. Expressions are literals, additions, and functions. Types are integer type and function (arrow) types.

```
<type> ::= "Int"
         | <type> "->" <type>

<expr> ::= literal
         | <expr> "+" <expr>
         | "\" ident ":" <type> "." <expr>
```

We definitely do not want to mix them up, otherwise we would have ill-formed terms such as `1 + Int`. Therefore, we use two extensible datatypes for them, the code below shows two visitor interfaces: `IType` for type ASTs and `IExpr` for expression ASTs. The `IAdd` is reused from Section 4.2.1.

```
trait IType[-R, E] {
  def int(): E
  def arrow(a: R, b: R): E
}
trait IExpr[-R, E, -T] extends IAdd[R, E] {
  def func(x: String, t: T, e: R): E
}
```

Because expressions contain types, we have an extra parameter `-T` in `IExpr`. As same as the recursive parameter `-R`, the minus annotation indicates contravariance. Thus, subtyping relationship is preserved, whichever of type and expression is extended.

However, writing the concrete datatypes as we shown in Section 4.2.2 is not straightforward. If we write the following synonyms using the `Exp` defined identical as before, a compile error will arise saying the number of type parameters is wrong.

```
trait Exp[-A[-R, _]] {
  def apply[E](vis: A[Exp[A], E]): E
}
type Type = Exp[IType]
type Expr = Exp[IExpr]   // Error! Number of parameters is not correct.
```

The problem is `IExpr` takes three type parameters while the abstract interface `A` in `Exp` expects only two. We cannot simply change `A` to take three parameters, since other components may require different numbers. Creating a new trait instead of `Exp` is not acceptable either, because we would lose subtyping relationship if so.

Fortunately, Scala allows us to write type level lambdas by type projections. We use the wrapper `Exp2` below to represent datatypes with two sorts.

```
type Exp2[-A[-R, E, -F], +V] = Exp[({type l[-R, E] = A[R, E, V]})#l]
```

The type parameter `V` is abstracted for the second sort. On the right hand side of the definition, the type lambda `l[-R, E]` which takes two parameters is passed to `Exp`. In the body of it, abstract interface `A` is partially applied, using `V` as the second sort.

Then we can rewrite the wrong code before to pass Scala's type checker.

```
type Type = Exp[IType]
type Expr = Exp2[IExpr, Type]
```

Because `Exp2` is just a type synonym, it does not introduce a new type identity. The `Expr` above is still derived from `Exp`, hence subtyping relationship is preserved. Namely, the type `Expr`, which equals `Exp2[IExpr, Type]`, is a supertype of `Exp[IAdd]`. As denoted by the plus sign of `V`, datatype of the second sort is covariant, hence it preserves subtyping as well.

The long type encoding of type lambdas is annoying, especially when writing operations. A compiler plugin called *kind-projector* [38] provides shorthand for such types.

With the plugin, we can have a quite concise definition for `Exp2` as below, where each question mark represents a type parameter.

```
type Exp2[-A[-R, E, -F], +V] = Exp[A[-?, ?, V]]
```

4.6 Context Propagation

In Section 4.2, we showed the modular external visitors enable us to write and reuse modular datatypes as well as operations (visitors). However, in practice, reusing operations could be more complicated than demonstrated before. Especially when modularizing languages and their interpreters that compute semantics, we often need to add auxiliary *contexts*. As a result, the mismatch of type signatures prevents new operations from extending and reusing existing ones.

This section first demonstrates this issue, then shows our solution which is inspired by *implicit context propagation* [30]. The crucial point is delegating recursive calls, based on the technique discussed in Section 4.4.

4.6.1 Motivation

Let us review the example in Section 4.2.4, in which we have `EvalSub` visitor extends and thus reuses `EvalAdd`. We need to point out there are two implicit conditions which make that extension possible. The first is that the `ISub` interface is a subtype of `IAdd`. The second is that the "return type" declared in the signatures of the two visitors are both `Int`.

The second condition, that new visitors must have exactly the same return type with the existing ones it intends to reuse, could be too restrictive in some cases. For instance, we consider extending the language of literals and additions by adding variables. The visitor interface could be easily extend as below.

```
trait IVar[-R, E] extends IAdd[R, E] {
  def vr(x: String): E
}
```

When evaluating the new language, we want to use an extra context which stores value bindings for variables. The context could be different in each evaluation procedure and should be passed as an argument. Therefore, the return type of the evaluation operation should be a function `Context => Int`.

In the code below, we are using a map for the context, and trying to reuse old operation `EvalAdd` by extending from it.

```
type Context = Map[String, Int]
trait EvalVar[A[-R, _]] extends IVar[Exp[A], Context => Int]
  with EvalAdd[A] {    // Illegal inheritance!
    ..
}
```

Unfortunately, the code will get a compile error of "Illegal inheritance", because in `EvalAdd` the return type is `Int` rather than `Context => Int`.

This issue is painful in terms of language modularization, because contexts are quite frequently used when writing interpreters or semantic functions. Basically, we have two options to make the code compile, but neither is satisfying.

- We can anticipate the future need of context at the very beginning. An abstract type parameter could be used to represent a context in the signature of `EvalAdd`. Their methods for every cases also take an extra context parameter, but do nothing with it except passing it down to subtrees.

  Anticipating future extensions does not actually solve the problem, but transfers the burden to early stages of the development. It would complicate the base operations by having unused and unnecessary contexts everywhere. Furthermore, it is not scalable for more than one context.

- We can implement all the methods such as `lit` and `add` again in the new operation `EvalVar`. Usually this is done by copying old code and modifying it.

  This workaround would lose the value of modularity, since no code is reused at all. Code which is almost the same would be duplicated in each extension and finally be hard to maintain.

4.6.2 Propagation by Delegating Recursive Calls

The vital issue of reusing existing operations is to pass the context downwards recursively. Because modification on old code is not acceptable, we can only pass the context implicitly. From previous comparison, we know that, in Object Algebras and internal visitors the values on recursive positions are results, while in external visitors they are subtrees of the datatype.

We first abstract recursive calls, as demonstrated in Section 4.4. New operations cannot inherit from existing ones because types are mismatched, but we can compose and incorporate their instances for reuse. When reusing existing operations, we delegate all recursive calls in them, in order to propagate the context without affecting their logic.

```
 1 trait IAdd[-R, E] {
 2   def lit(x: Int): E
 3   def add(l: R, r: R): E
 4   def apply(x: R): E
 5 }
 6 trait EvalAdd[A[-R, _]] extends IAdd[Exp[A], Int] {
 7   def lit(x: Int): Int = x
 8   def add(l: Exp[A], r: Exp[A]): Int = apply(l) + apply(r)
 9 }
10 trait IVar[-R, E] extends IAdd[R, E] {
11   def vr(x: String): E
12 }
13 type Context = Map[String, Int]
14 trait EvalVar[A[-R, _]] extends IVar[Exp[A], Context => Int] {
15   def propagate(c: Context) = new EvalAdd[A] {
16     def apply(x: Exp[A]): Int = EvalVar.this.apply(x)(c)
17   }
18   def vr(x: String): (Context) => Int = _ (x)
19   override def lit(x: Int): (Context) => Int = propagate(_).lit(x)
20   override def add(l: Exp[A], r: Exp[A]): (Context) => Int =
21     propagate(_).add(l, r)
22 }
```

FIGURE 4.1: An example of context propagation by delegating recursive calls.

Figure 4.1 shows an example of context propagation using our approach. It is the example we used in the last section, adding variables to a language of literals and additions. Lines 1 to 12 are visitor interfaces, and the old evaluation operation `EvalAdd` whose return type is just `Int`. From line 14 we create a new operation `EvalVar` whose return type is `Context => Int`.

In `EvalVar`, three methods `vr`, `lit`, and `add` are implemented. The last two are old methods we intend to reuse from `EvalAdd`. They are overridden by feeding the context to the `propagate` method, which is the crucial point. `propagate` creates instances of `EvalAdd` and implements the `apply` method for recursive calls. The `apply` of `EvalAdd` is delegated to the `apply` of `EvalVar`, which has the context added thus it can be propagated to subtrees.

Consequently, the evaluation operation is extended with a context while retaining semantic modularity. This approach is inspired greatly by *implicit context propagation* [30], which was originally proposed for Object Algebras. However, it is quite different since we are working on external visitors rather than internal visitors and Object Algebras.

4.6.3 Lifter

Although the context propagation approach we demonstrated works well, we still have some redundant code in Figure 4.1. Specifically, the two methods `lit` and `add` in `EvalVar` are overridden by hand, and they do nothing but call `propagate`.

If we have many such methods in old operations, it will be painful for the user to override all of them. Moreover, they would be duplicated if there were more than one new operations extending from a same existing operation.

Fortunately, we can abstract those boilerplate code as a template, we call it a *lifter*. Every visitor interface has its own lifter, for adding a context parameter to existing operations. The code below shows lifter for interface IAdd in the previous example.

```
trait Lifter[-R, E, C] extends IAdd[R, C => E] {
  def propagate(c: C): IAdd[R, E]
  def lit(x: Int): C => E = propagate(_).lit(x)
  def add(l: R, r: R): C => E = propagate(_).add(l, r)
}
```

It has an abstract type parameter C for context. Operations whose return type is E are transformed as operations of type C => E. With this lifter, we can easily rewrite the EvalVar in Figure 4.1 as below.

```
trait EvalVar[A[-R, _]] extends IVar[Exp[A], Context => Int]
  with Lifter[Exp[A], Int, Context] {
  def propagate(c: Context): IAdd[Exp[A], Int] =
    new EvalAdd[A] {
      def apply(x: Exp[A]): Int = EvalVar.this.apply(x)(c)
    }
  def vr(x: String): (Context) => Int = _ (x)
}
```

Now we only need to implement the new method vr for variables. All old ones are handled automatically by extending from Lifter.

Furthermore, lifters have two advantages worth mentioning.

- They are modular. Because they are actually concrete visitors, they can be composed modularly along with corresponding visitor interfaces, and only deal with new added methods.
- They can be generated. The pattern of lifters is clear, that they declare an abstract method propagate and delegate all methods to it. Therefore, they can be generated automatically from visitor interfaces. Later in Section 6.3, we will discuss code generation including lifters.

4.6.4 Disscussion

Adding contexts is a special case of managing effects. In functional programming, effects are usually encapsulated by monads [42, 65]. Our context propagation is an analog to the reader monad if we exclude mutable states. In terms of language implementation, many typing semantics only involve read-only contexts, and thus context propagation can be useful. The pattern is able to cover more situations with mutable states, as demonstrated in the implicit context propagation[30] work.

A more elegant way to resolve the mismatching of return types is to return monadic values and keep the monad abstract until the final instantiation. With a proper encoding of monad, monad transfers [40], and MTL-style type class constraints [32, 26], the previous example could be implemented like below:

```
trait EvalAdd[A[-R, _], M[_]: Monad] extends IAdd[Exp[A], M[Int]] {...}
trait EvalVar[A[-R, _], M[_]: MonadReader] extends IVar[Exp[A], M[Int]] with
    EvalAdd[A, M] {...}
```

That is actually our initial attempt to address the problem. However, we failed to build such a pattern due to issues [12, 57] of the current monad encoding using Scala's implicits. The issues prevents us from using MTL-style type class constraints as in Haskell, and it may require language changes to Scala to fix them. We hope a future release of Scala will support better monad encodings and then makes this solution possible.

4.7 Case Analysis on ASTs

When implementing operations, it is very common to analyse datatypes (and ASTs) case by case and process them differently for each case. *Pattern matching* is a neat solution for such analysis and it is closely related to visitors. Scala has pattern matching on case classes and a `sealed` modifier to enable the compiler to check exhaustiveness.

However, datatypes should not be `sealed` in the modular setting, thus exhaustiveness in pattern matching has to be sacrificed if we use case classes. Moreover, case classes are deeply rooted in the type system of Scala. If a set of case classes are not `sealed`, it cannot be implemented using traditional OO structures in a type safe manner. That would obstruct the migration of code to other OO languages which do not have such mechanisms.

This section presents our solution for analyzing datatypes with static type safety. Our approach uses only visitors, thus exhaustiveness is guaranteed. The key idea is to abstract the top level representation of datatypes and concretize it when needed.

4.7.1 Motivation

To illustrate the issue, we use an example of boolean literals and if-then-else expressions. The visitor interface is written below.

```
1  trait Eval[A[-X, Y] <: IBool[X, Y]] extends IBool[Exp[A], Exp[A]] {
2    val isVal: A[Exp[A], Boolean]
3    def tTrue(): Exp[A] = TTrue[A]()
4    def tFalse(): Exp[A] = TFalse[A]()
5    def tIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
6      if (e1(isVal))
7        // if e1 is true/false then e2/e3, else raise an error
8        ???
9      else
10       TIf(apply(e1), e2, e3)
11 }
```

FIGURE 4.2: An evaluation operation which requires analysis on a subtree.

```
  trait IBool[-R, E] {
    def tTrue(): E
    def tFalse(): E
    def tIf(e1: R, e2: R, e3: R): E
    def apply(e: R): E
  }
```

Three corresponding factory classes `TTrue`, `TFalse` and `TIf` are implemented using the pattern shown in Section 4.2.3. We omit their code here.

Figure 4.2 shows an incomplete implementation of small step evaluation. The two cases for boolean literals are trivial. For the if-then-else case, we first check whether the condition expression is a value. If it has not been reduced to a value, we apply the evaluation operation on it recursively. Otherwise, we need to perform a case analysis on it to decide which branch to return, or raise an error if the value is neither true nor false.

Now let us consider how to fill the blank at line 7. A straightforward way is to use case classes and pattern matching.

```
e1 match {
  case TTrue() => e2
  case TFalse() => e3
  case _ => sys.error("Error")
}
```

As discussed before, the exhaustiveness checking is sacrificed and the use of case classes makes the implementation heavily relies on the language features of Scala. The code cannot be written using traditional OO language features with static type safety. Furthermore, it is awkward to use visitors outside and case classes inside, regarding the fact they are closely related.

Unfortunately, it is not easy to employ visitors in this situation. The subtree `e1` we want to analyse has an abstract type `Exp[A]`, where `A` is a subtype of the visitor interface `IBool`. That makes it impossible to create a visitor and feed it to `e1`. The following code shows our failed attempt.

```
e1(new A[Exp[A], Exp[A]] {   // Illegal, because A is abstract
  ...
})
```

The problem is that some parts of A are unknown at the current time. Although they are irrelevant to the analysis, we still have to specify the corresponding results.

4.7.2 Abstracting the Top Level Interface

The abstraction of visitor interface A is essential to extensibility, but now it blocks the creation of visitors. The key of our solution is to separate the top level visitor interface and the interface used in subtrees.

The Exp trait we used previously was defined as following.

```
trait Exp[-A[-R, _]] {
  def apply[E](vis: A[Exp[A], E]): E
}
```

A datatype Exp[A] is represented by a function which expects visitors of shape A, and all subtrees have the same type of Exp[A].

However, it is not necessary to use a same interface A in both places. Instead, we can abstract the interface at the top level as A, and the interface used in subtrees as B.

```
trait SExp[-A[-R, _], -B[-F, _]] {
  def apply[E](alg: A[SExp[B, B], E]): E
}
```

Compatibility with the original Exp is easily preserved using a type synonym, which sets both A and B to be the same again.

```
type Exp[-A[-R, _]] = SExp[A, A]
```

This encoding requires some small modifications on the factory classes. A new type parameter needs to be added and the return type needs to be changed. We demonstrate a modified factory TTrue for the boolean true case as an example.

```
case class TTrue[A[-X, Y] <: IBool[X, Y], B[-R, _]]() extends SExp[A, B] {
  def apply[E](vis: A[Exp[B], E]): E = vis.tTrue()
}
```

The new representation SExp allows us to specify the type of ASTs more clearly, and enables type conversions of subtrees. After conversions, the top level interface becomes concrete, therefore visitors can be applied.

Figure 4.3 show our implementation of the evaluation operation, using visitors for analyzing the condition subtree in the if-then-else case. The auxiliary trait Convert

```
1  trait Convert[A[-X, Y] <: IBool[X, Y]] {
2    def convert[B[-R, _]](e: SExp[A, B]): Option[SExp[IBool, B]]
3  }
4  trait Eval[A[-X, Y] <: IBool[X, Y]] extends IBool[Exp[A], Exp[A]]
5    with Convert[A] {
6    ...
7    def tIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
8      if (e1(isVal)) {
9        val c: SExp[IBool, A] =
10         convert(e1).getOrElse(sys.error("Conversion failed"))
11       c(new IBool[Exp[A], Exp[A]] {
12         def tTrue(): Exp[A] = e2
13         def tFalse(): Exp[A] = e3
14         def tIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
15           sys.error("Not a value")
16         def apply(e: Exp[A]): Exp[A] = sys.error("Impossible")
17       })
18     }
19     ...
20 }
```

FIGURE 4.3: Type conversion of a subtree and analysis using visitors.

contains a generic method `convert`. The method converts the type of the parameter `e` from `SExp[A, B]` to `SExp[IBool, B]` for arbitrary `B`, as long as `A` is a subtype of `IBool`. Possible failure of the conversion is indicated by the `Option` type. In the evaluation, `convert` is used in line 10. The type of the subtree `e1` is converted to `SExp[IBool, A]`. Therefore, a visitor of type `IBool[Exp[A], Exp[A]]` can be applied for implementing the semantics of if-then-else, as demonstrated by line 11 to 17.

4.7.3 Conversion and Chaining

Now the problem becomes how to implement the method for type conversion. If we use `IBool` as a language directly, the method will be the injection function of `Option`. Except this trivial case, we need to separate new variants and old ones in `IBool` after extensions.

We first define an injection operation which just wraps an expression in `Option`. The following code shows an implementation for the `IBool` interface.

```
trait Inject[A[-X, Y] <: IBool[X, Y], B[-R, _]]
  extends IBool[Exp[B], Option[SExp[A, B]]] {
  def tTrue(): Option[SExp[A, B]] = Some(TTrue[A, B]())
  def tFalse(): Option[SExp[A, B]] = Some(TTrue[A, B]())
  def tIf(e1: Exp[B], e2: Exp[B], e3: Exp[B]): Option[SExp[A, B]] =
    Some(TIf[A, B](e1, e2, e3))
}
```

```
trait ConvertChain[A[-X, Y] <: IExtBool[X, Y]]
  extends Convert[A] with ConvertExt[A] {
  def convert[B[-R, _]](e: SExp[A, B]): Option[SExp[IBool, B]] = {
    val v = new IExtBool[Exp[B], Option[SExp[IBool, B]]] with Inject[IBool, B] {
      def not(e: Exp[B]): Option[SExp[IBool, B]] = None
      def apply(e: Exp[B]): Option[SExp[IBool, B]] = sys.error("Impossible")
    }
    convertExt(e).flatMap(_ (v))
  }
}
```

FIGURE 4.4: An example of chaining conversions.

Then let us consider extensions. For instance, we add the boolean operator "not" to the language by creating a new visitor interface `IExtBool` which extends `IBool`.

```
trait IExtBool[-R, E] extends IBool[R, E] {
  def not(e: R): E
}
```

A new conversion which restricts the top level interface to be `IExtBool` is introduced correspondingly.

```
trait ConvertExt[A[-X, Y] <: IExtBool[X, Y]] {
  def convertExt[B[-R, _]](e: SExp[A, B]): Option[SExp[IExtBool, B]]
}
```

We have two conversions at present and they have a strong connection, that is. their results have a subtyping relationship. Consequently, we can use `convertExt` to implement `convert`. Figure 4.4 shows such chaining of the two conversions.

Chaining of conversions has advantages comparing with implementing them individually.

Firstly, it reduces method instantiations when finalizing operations as objects. For example, the evaluation of the new interface `EvalExt` could be implemented using `ConvertChain` as following.

```
trait EvalExt[A[-X, Y] <: IExtBool[X, Y]] extends IExtBool[Exp[A], Exp[A]]
  with Eval[A] with ConvertChain[A] {
  def not(e: Exp[A]): Exp[A] = ...
}
object EvalExt extends EvalExt[IExtBool] {
  val isVal: IExtBool[Exp[IExtBool], Boolean] = ...
  def convertExt[B[-R, _]](e: SExp[IExtBool, B]): Option[SExp[IExtBool, B]] =
    Some(e)
  def apply(e: Exp[IExtBool]): Exp[IExtBool] = e(this)
}
```

Both of the two conversions are available in the trait `EvalExt`, whereas only the new `convertExt` needs to be instantiated in the object `EvalExt`. Even after several extensions, the number of undefined conversion is still one, which is the conversion to the current interface.

Secondly, it preserves the property of maintaining only direct dependencies between language components (see Section 6.2). In other words, old conversions are hidden and dependencies on them are transformed to new ones in extensions. Furthermore, it enables local code generation for automatically creating conversions, as well as the injection and chaining utilities. As a result, users do not need to implement any auxiliary structures by themselves. Section 6.3 discusses details about code generation.

To summarize, our solution for analyzing ASTs uses pure visitors, with both extensibility and exhaustiveness. It does not depends on any language specific features, and it brings little overhead to users since conversions and chaining could be automatized. Even if we wrote them by hands, it would not be hard to obtain correct implementations because main structures are checked with visitor interfaces.

Chapter 5

Type-safe Modular Parsing[1]

In previous chapters we showed modularization of ASTs and operations. This chapter presents our work for type-safe modular parsing, that allows parsers to evolve together with abstract syntax. Our approach achieves semantic modularity, therefore parsers can be modularly type-checked and separately compiled. In Section 5.1 we introduce the requirements of modular parsing problem. In Section 5.2 we discuss the choice of parsing techniques. In Section 5.3 we show our approach of type-safe modular parsing using traditional OO ASTs. To illustrate the generality of the technique and achieve greater modularity, we show variants of the technique using Object Algebras in Section 5.4, and modular external visitors in Section 5.5. The last variant using modular external visitors is used in our framework Gems.

5.1 Modular Parsing Problem

Operations such as evaluation and pretty-printing which *process* or *consume* ASTs are important for implementing programming languages, and we have shown how to modularize them. However, operations that *produce* ASTs should not be ignored, since programmers normally do not write ASTs by hand. Parsing is one such operation, which is fundamental as the "front end" of interpreters and compilers.

Parsing is the bridge between *concrete syntax* and *abstract syntax* of a language. Usually both of them have clear specifications that implementations must follow. Therefore, parsing should be incorporated when modularizing languages as components.

We prefer semantic modularity for parsers. To be concrete, we pose modular parsing problem by listing requirements as follows:

- **Extensibility and reusability**: When abstract syntax evolves, parsers should be able to be extended and reused. Identical code should not be duplicated, and existing code should not be modified.

[1]As stated in the acknowledgements, this chapter describes joint work [70] with Haoyuan Zhang.

- **Separate compilation and type-checking**: Parsers should be modularly type-checked and separately compiled. The parsing technique is not allowed to access source code, grammar definition, or other textual content of existing parsers.
- **Independent from abstract syntax**: Having multiple parsers for one same set of abstract syntax should be supported. Thus, the boundary of concrete and abstract syntax is clear.

Futhermore, if ASTs can be composed in a independent, non-linear manner, corresponding parsers should also be able to do so.

These requirements rule out nearly all syntactically modular approaches for extensible parsing [25, 24, 58], because global generation and analysis are forbidden. To fulfill the requirements, we need to solve two major challenges.

The first challenge is how to choose proper parsing techniques. In Section 5.2, we will discuss it in details. The second challenge is how to define, extend and compose parsers. We will show general approaches in Section 5.3, and demonstrate variants in Section 5.4 and 5.5, since it varies on different representations of AST.

5.2 Choose Parsing Techniques for Modularity

Although a lot of parsing techniques and algorithms have been developed, they are usually not designed with concern for modularity and extensibility. Especially, parser generators such as the famous tool Yacc [31] have difficulties in extending parsers.

One reason is that parser generators often require *full* information about the syntax for generating parsers. Once the syntax is changed, even a little, the global generation must be performed again. Futhermore, in practice we usually have source code fragments attached with the syntax, as actions for building ASTs during parsing. That code is only glued and type-checked *after* the generation, thus static type safety is not assured. Such deficiencies are against the requirements of the modular parsing problem.

As discussed in Section 2.5, parser combinators are closer to the host language and more flexible regarding composition, that makes them quite suitable for modular parsing. Parsers are naturally extensible, since they can be combined and manipulated by programmers. Additionally, they can be statically and separately type-checked, because the code are written directly in the host language rather than generated by tools.

5.2.1 Challenges with Parser Combinators

However, even with parser combinators, there are still several difficulties towards modular setting. Ordinary parser combinator libraries including Haskell's Parsec [39] do not fulfill modularity requirements, thus cannot be directly adopted. In particular, left-recursion, priority of alternatives, and backtracking are three major blocking issues. Those issues are discussed next.

Left-recursion Most parser combinator libraries employ the top-down, recursive descent parsing strategy, which cannot support left-recursive grammars directly. We use the example of literals and addition expressions again, the syntax is below.

```
<exp> ::= literal
        | <exp> "+" <exp>
```

Notice that the addition branch in the syntax is left-recursive. The following code shows parsers written by directly following the syntax, using Parsec.

```
parseExp = parseAdd <|> parseLit
parseAdd = do
  e <- parseExp
  ...
```

This implementation will get into an infinite loop, because when running the parser, `parseExp` and `parseAdd` call each other and never stop.

A common solution for this issue is to transform the syntax into an equivalent but not left-recursive one. This is called *left-recursion elimination*. For our example, a transformed syntax is shown below.

```
<exp>  ::= literal <exp'>
<exp'> ::= empty
         | "+" <exp'>
```

After left-recursion elimination, the structure of grammar is changed, as well as its corresponding parser. Since the access to textual content of existing components is forbidden, it is very complicated to analyse and rewrite the grammar when doing extensions. Reusing the existing compiled parsing code is also hard after the transformation.

A workaround is anticipating that every non-terminal is left-recursive, regardless its current productions, at the very beginning of development. However, this is quite cumbersome and overkill, because it unnecessarily pollutes syntax representation and parser implementation.

Moreover, left-recursion elimination requires extra bookkeeping to retain the original parsing trees. Actions for building ASTs are often tied closely with parsing trees, hence it is quite hard to reuse them.

Priority of Alternatives Considering the alternative definitions (interpreted as "or") in syntax, the order or priority of them should be managed carefully in practice. As an example, we could have two alternative branches and one is a prefix of the other.

```
<exp> ::= literal
        | literal "+" <exp>
```

Following the syntax, its parser is possibly written as:

```
parseExp = parseLit <|> parseAdd
parseAdd = do
  x <- parseLit
  ...
```

However, a subtle issue is that Parsec's alternative combinator `<|>` will try all the alternative parsers in the given order, and return the first successful parsing result, even if it only consumes a prefix of the input and subsequent parsers may parse the whole input. Specifically, `parseExp` parses input `"1 + 2"` as `1`, because the `parseLit` successfully consumes `1` hence it will not try the next branch.

Regarding extensibility, the correctness of parsers heavily depends on the strategy of alternative combinator. One approach to resolve the problem in most cases is to use a *longest match* alternative combinator, which selects the longest parsing result from all successful ones. For more complex cases, a *prioritized match* alternative combinator could be employed, and every alternative branch should be attached with a comparable priority value provided by the user. Therefore, the order of selecting results is explicitly under control.

Backtracking Backtracking is another blocking issue related with alternative processing. Among all the alternative branches, if any two of them share a common prefix, the parser must backtrack to obtain the correct result.

```
<stmt> ::= "import" ident "from" ident
         | "import" ident "as" ident
```

Considering the syntax above, if the first "import-from" alternative fails after consuming the "import" keyword and the identifier, we must backtrack to the beginning and start from "import" again to try the second alternative.

Given the full syntax, we can decide when to backtrack. Usually we only do so when necessary to achieve better performance. For example, backtracking is off by default in Parsec and programmers need to use a `try` function to backtrack explicitly. However, in the modular setting we must always backtrack, because we do not know what will be added in the future. That results in the worst-case exponential time complexity, and thus is not satisfying.

```
 1 import scala.util.parsing.combinator.PackratParsers
 2 import scala.util.parsing.combinator.syntactical.StandardTokenParsers
 3 
 4 object Parser extends StandardTokenParsers with PackratParsers {
 5   lexical.delimiters += ("(", ")", "+")
 6 
 7   lazy val pLit: PackratParser[Int] = numericLit ^^ { _.toInt }
 8   lazy val pExp: PackratParser[Int] =
 9     pLit |||
10     "(" ~> pExp <~ ")" |||
11     pExp ~ ("+" ~> pExp) ^^ { case x ~ y => x + y }
12 
13   def parse(input: String): Int =
14     phrase(pExp)(new lexical.Scanner(input)).get
15 }
```

FIGURE 5.1: An example of using Packrat parsing in Scala.

5.2.2 Packrat Parsing

To resolve aforementioned issues in modular parsing, we need more powerful techniques than ordinary parser combinators. Packrat parsing [19] is a good choice. It uses a memoization table to store the results of applying each parser at each position of the input, so that repeated parsing is eliminated. As a result, full backtracking does not cost exponential time but only linear time for non-left-recursive grammars. Futhermore, Packrat parsing can be extended to support both direct and indirect left-recursive grammars [68]. It is therefore very suitable as the underlying parsing technique for building modular parsers.

In Section 2.5 we introduced Scala's parser combinator library. The library includes Packrat parsing utilities with direct left recursion support, and the longest match alternative combinator. In Figure 2.4, the parser is not written in a left-recursive manner. We rewrite it using Packrat parsing and the longest match alternative combinator in Figure 5.1.

In the code, we just change the return type to `PackratParser`, which is inherited from the `PackratParsers` trait. Then the parser `pExp` can be written in a left-recursive way. From line 9 to line 11, the three alternative branches are composed by the longest match combinator `|||`, which guarantees all of them will be applied. Readers may refer to Table 2.1 for description of parser combinators.

5.3 Modular Parsing for OO ASTs

With Packrat parsing as a nice underlying parsing technique, now we can focus on the modularity issue itself. A starting point is to use traditional object-oriented ASTs, which often consist of an abstract class as the datatype, and several subclasses

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

trait BaseParser extends StandardTokenParsers with PackratParsers {
  def parse[T](parser: PackratParser[T])(input: String): T =
    phrase(parser)(new lexical.Scanner(input)).get
}
```

FIGURE 5.2: An auxiliary parser trait for code demonstration.

as different data variants. Adding new data variants is easy using OO ASTs and semantic modularity is preserved naturally.

We will use a simple example in this and the following sections, that extending a language of literals and addition expressions by adding variables as a new data variant. The syntax is shown below.

```
<exp> ::= literal
        | <exp> "+" <exp>
        | ident
```

Initial ASTs and Parser It is straightforward to write corresponding classes for ASTs before the extension. We add a simple operation `print` to them.

```
trait Exp {
  def print: String
}
case class Lit(x: Int) extends Exp {
  def print: String = x.toString
}
case class Add(l: Exp, r: Exp) extends Exp {
  def print: String = s"${l.print} + ${r.print}"
}
```

For better demonstration of parsing code, we define an auxiliary trait `BaseParser` in Figure 5.2. The parser before extension is implemented as follow.

```
trait Parser extends BaseParser {
  lexical.delimiters += ("(", ")", "+")
  lazy val pLit: PackratParser[Exp] =
    numericLit ^^ { x => Lit(x.toInt) }
  lazy val pAdd: PackratParser[Exp] =
    pExp ~ ("+" ~> pExp) ^^ { case l ~ r => Add(l, r) }
  lazy val pExp: PackratParser[Exp] =
    pLit ||| pAdd ||| "(" ~> pExp <~ ")"
}
```

Extending the ASTs and Parser Extending the abstract syntax only requires adding a new class which implements the `Exp` interface.

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

Considering extension of the parser, inheriting from the old one is essential to reuse code. An attempt of writing the new parser is shown below. A new `pVar` is added for variables, and expressions are parsed by `pExp ||| pVar`.

```
trait NewParser extends Parser {
  lazy val pVar: PackratParser[Exp] = ident ^^ Var
  // Wrong
  lazy val pNewExp: PackratParser[Exp] = pExp ||| pVar
}
```

However, this attempt fails to parse valid input like `"1 + x"`. The reason is that the old parser contains recursive calls of `pExp` for parsing sub-expressions, but those recursive calls are not updated to include the new added `pVar`.

Therefore, we must override the inherited `pExp` to incorporate `pVar`. This is the key point in our approach of writing modular parsers.

```
trait NewParser extends Parser {
  lazy val pVar: PackratParser[Exp] = ident ^^ Var
  // Correct but cannot compile
  override lazy val pExp: PackratParser[Exp] = super.pExp ||| pVar
}
```

Because of dynamic dispatch, all recursive calls of `pExp` in the inherited code are properly updated. However, as remarked in the comment, there is a subtle issue of Scala that blocks the code to be compiled. Scala's `super` keyword cannot refer to fields that declared by the `val` keyword, but the parser combinator library requires the use of `lazy val` for left recursion. The incompatibility of `super` and `val` is a bug of Scala for years [53], and it has not been fixed for unknown reasons.

Explicit Name Resolution The problem is that the old field is shadowed by the new one when overriding it. A workaround is to make an explicit copy of the field with a distinct name, thus it can be referred unambiguously. The initial parser could be written as below.

```
trait Parser extends BaseParser {
  lexical.delimiters += ("(", ")", "+")
  lazy val pLit: PackratParser[Exp] = numericLit ^^ { x => Lit(x.toInt) }
  lazy val pLitAdd: PackratParser[Exp] =
    pLit |||
    "(" ~> pExp <~ ")" |||
    pExp ~ ("+" ~> pExp) ^^ { case l ~ r => Add(l, r) }
  lazy val pExp: PackratParser[Exp] = pLitAdd
}
```

The `pLitAdd` and `pExp` are identical. That allows us to use `pLitAdd` to represent `pExp` in the extended parser.

```
trait NewParser extends Parser {
  lazy val pVar: PackratParser[Exp] = ident ^^ Var
  lazy val pLitAddVar: PackratParser[Exp] = pLitAdd ||| pVar
  override lazy val pExp: PackratParser[Exp] = pLitAddVar
}
```

In the new extended parser, a new name `pLitAddVar` is introduced as a copy of `pExp`, so that we are able to use it afterwards in the same manner.

This workaround is lightweight, however it still pollutes the name space. The problem is not fundamental but quite specific to Scala and the parsing library. It could be avoided in other languages which have better support for the "super" reference.

Independent Extensibility Instead of extending parsers linearly, we can compose several parsers at the same time using multiple inheritance to achieve independent extensibility.

If we ignore the aforementioned name shadowing issue, two independent parsers from `LanguageA` and `LanguageB` can be composed as below.

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

5.4 Modular Parsing with Object Algebras

In Section 2.4, we showed that Object Algebras are a lightweight solution to the expression problem. Comparing with OO ASTs, Object Algebras bring extensibility to not only data variants but also operations. The modular parsing pattern demonstrated in the last section can be adopted to work with Object Algebras.

Initial Parser Using the same example in the last section, we write Object Algebra interface of the initial language as below.

```
trait IAdd[E] {
  def lit(x: Int): E
  def add(l: E, r: E): E
}
```

The corresponding parser contains a field of the Object Algebra interface. Auxiliary trait `BaseParser` is defined in Figure 5.2.

```
trait Parser[E] extends BaseParser {
  val alg: IAdd[E]
  lexical.delimiters += ("(", ")", "+")
  lazy val pLit: PackratParser[E] =
    numericLit ^^ { x => alg.lit(x.toInt) }
  lazy val pAdd: PackratParser[E] =
    pExp ~ ("+" ~> pExp) ^^ { case l ~ r => alg.add(l, r) }
  lazy val pLitAdd: PackratParser[E] =
    pLit ||| pAdd ||| "(" ~> pExp <~ ")"
  lazy val pExp: PackratParser[E] = pLitAdd
}
```

In the code above, the `alg` field provides methods for parsers to build results. It is at a covariant position, thus can be properly overridden and refined afterwards.

Extending the Parser Considering the extension of adding variables, the Object Algebra interface is extended in the standard way.

```
trait IVar[E] extends IAdd[E] {
  def vr(x: String): E
}
```

The parser is extended similarly as before. The type of the Object Algebra field `alg` is refined to be `IVar[E]`, so that the method `vr` can be used for the variable case.

```
trait NewParser[E] extends Parser[E] {
  override val alg: IVar[E]
  lazy val pVar: PackratParser[E] = ident ^^ alg.vr
  lazy val pLitAddVar: PackratParser[E] = pLitAdd ||| pVar
  override lazy val pExp: PackratParser[E] = pLitAddVar
}
```

Independent extensibility of parsers is supported using multiple inheritance, as discussed in the last section.

A Deficiency Object Algebras are more flexible that traditional OO ASTs. However, a deficiency is that we do not have a concrete representation of ASTs. That makes parsing and operations entangled in our examples, therefore each operation performs an reparsing of the input. For example, client code using `NewParser` is shown below. It actually involves two processes: parsing from input and applying the `Print` operation.

```
trait Print extends IVar[String] {...}
val parser = new NewParser[String] {
  val alg: IVar[String] = new Print {}
}
val s: String = parser.parse(parser.pExp)("1 + 2")
```

The result is a string rather than an AST. If we want to apply another operation, the parser must be created again to instantiate a different `alg` field.

This is a known problem [24]. Existing solutions include merging operations together [52], and using reflection to record the structure [24]. Neither one is completely satisfying in terms of semantic modularity. To actually solve the problem, we need a concrete representation of ASTs such as modular external visitors.

5.5 Modular Parsing for Visitor ASTs

In Chapter 4, we discussed modular external visitors. They are heavier than Object Algebras with regard to encoding, but provide more functionality, including different type identities, subtyping relationships, and explicit traversal control. Most importantly, modular visitors have a concrete representation of ASTs which can be naturally used as parsing results.

We continue using the previous example for demonstration.

Initial ASTs and Parser We write the visitor interface for the initial language as below. Readers may refer to Section 4.2 for details of modular external visitors.

```
trait IAdd[-R, E] {
  def lit(x: Int): E
  def add(l: R, r: R): E
  def apply(x: R): E
}
```

Two factories `Lit` and `Add` are defined correspondingly for building concrete ASTs.

```
case class Lit[A[-X, Y] <: IAdd[X, Y]](x: Int) extends Exp[A] {
  def apply[E](vis: A[Exp[A], E]): E = vis.lit(x)
}
case class Add[A[-X, Y] <: IAdd[X, Y]](e1: Exp[A], e2: Exp[A]) extends Exp[A] {
  def apply[E](vis: A[Exp[A], E]): E = vis.add(e1, e2)
}
```

The parser is defined as follow using the auxiliary trait `BaseParser` in Figure 5.2.

```
trait Parser[A[-X, Y] <: IAdd[X, Y]] extends BaseParser {
  lexical.delimiters += ("(", ")", "+")
  lazy val pLit: PackratParser[Exp[A]] =
    numericLit ^^ { x => Lit[A](x.toInt) }
  lazy val pAdd: PackratParser[Exp[A]] =
    pExp ~ ("+" ~> pExp) ^^ { case l ~ r => Add(l, r) }
  lazy val pLitAdd: PackratParser[Exp[A]] =
    pLit ||| "(" ~> pExp <~ ")" ||| pAdd
  lazy val pExp: PackratParser[Exp[A]] = pLitAdd
}
```

The parser trait has a type parameter `A` which is restricted to be a subtype of `IAdd`. That makes `Exp[A]` to be a supertype of `Exp[IAdd]`, therefore `Lit` and `Add` are valid data variants as parsing results.

Extending the ASTs and Parser We first extend the visitor interface and add a new factory class for the new case.

```
trait IVar[-R, E] extends IAdd[R, E] {
  def vr(x: String): E
}
case class Var[A[-X, Y] <: IVar[X, Y]](x: String) extends Exp[A] {
  def apply[E](vis: A[Exp[A], E]): E = vis.vr(x)
}
```

The new parser is written as below. It is very similar to the one in Section 5.3.

```
trait NewParser[A[-X, Y] <: IVar[X, Y]] extends Parser[A] {
  lazy val pVar: PackratParser[Exp[A]] = ident ^^ Var[A]
  lazy val pLitAddVar: PackratParser[Exp[A]] = pLitAdd ||| pVar
  override lazy val pExp: PackratParser[Exp[A]] = pLitAddVar
}
```

In order to include the new case, the upper bound of type parameter `A` is refined to the new interface `IVar`.

Apart from the type parameter, the use of modular external visitors does not introduce any complexity to the parser comparing with OO ASTs. Parsing results have a concrete AST representation, therefore they can be used flexibly and following operations are independent of parsing. The combination of semantically modular parsing and techniques presented in Chapter 4 enables us to have a fully extensible framework of language implementation.

Chapter 6

Modular Language Components

In Chapter 3 we showed examples of using our framework Gems to modularize languages. Such modularization enables us to abstract features from languages as components and reuse them. This chapter discusses more details about language components. In Section 6.1 we introduce the idea of packing related structures together to abstract common language features. In Section 6.2 we show how to compose language components, considering only direct dependencies among them. In Section 6.3 we use metaprogramming to automatically generate boilerplate code in language components so that users do not bother to write them by hand.

6.1 Language Components

The modularization techniques we demonstrated in previous chapters enable us to abstract language features as reuseable, independent components. For example, most programming languages have boolean literals, logical operators, and if-then-else expressions. They can be packed into a component. When a language includes the component, its syntax is extended to incorporate those structures and it immediately knows how to parse, traverse, and manipulate them.

It can be very useful for rapid development of a Domain Specific Language (DSL), because we do not need to design and implement all of the language from scratch. Instead we can reuse existing language components. Futhermore, all techniques we introduced support semantic modularity, that guarantees type safety and efficiency of language composition.

A language component may contain the following structures.

- **Abstract syntax**: Abstract syntax structures are the core of a language component. We use modular external visitors (Chapter 4) to represent them.
- **Parsers**: Corresponding parsers are often needed to build ASTs. We use the modular parsing pattern described in Chapter 5 to modularize them.

```
object Common {
  trait SExp[-A[-R, _], -B[-F, _]] {
    def apply[E](vis: A[SExp[B, B], E]): E
  }
  type Exp[-A[-R, _]] = SExp[A, A]
  trait BaseParser extends StandardTokenParsers with PackratParsers {
    def parse[T](parser: PackratParser[T])(input: String): T =
      phrase(parser)(new lexical.Scanner(input)).get
  }
  trait Default[T] {
    def default: T
  }
}
```

FIGURE 6.1: Auxiliary structures shared in language components.

```
package component.bool
...
trait Term[-R, E] {
  def tmTrue(): E
  def tmFalse(): E
  def tmIf(e1: R, e2: R, e3: R): E
  def apply(e: R): E
}
case class TmTrue[A[-X, Y] <: Term[X, Y], B[-R, _]]() extends SExp[A, B] {
  def apply[E](vis: A[Exp[B], E]): E = vis.tmTrue()
}
case class TmFalse[A[-X, Y] <: Term[X, Y], B[-R, _]]() extends SExp[A, B] {
  def apply[E](vis: A[Exp[B], E]): E = vis.tmFalse()
}
case class TmIf[A[-X, Y] <: Term[X, Y], B[-R, _]](e1: Exp[B], e2: Exp[B], e3:
    Exp[B]) extends SExp[A, B] {
  def apply[E](vis: A[Exp[B], E]): E = vis.tmIf(e1, e2, e3)
}
```

FIGURE 6.2: Abstract syntax and factories in a language component.

- **Operations**: Concrete operations such as evaluation and pretty-printing can be implemented by modular external visitors as well.
- **Traversal templates**: Operations often involve traversal of ASTs. The *Shy* [71] framework proposed several traversal patterns to eliminate boilerplate code. They could be incorporated for implementing operations.
- **Other utilities**: Other utilities can be included as needed. Especially, lifters [30] for context propagation (Section 4.6), and type conversions (Section 4.7) are useful in practice.

For instance, we show a simple language component consisting of boolean literals and if-then-else expressions. In Figure 6.1 we define some auxiliary structures which are shared in language components. The abstract syntax and factories are defined in Figure 6.2.

```
package component.bool
...
trait Query[-R, E] extends Term[R, E] with Default[E] {
  def tmTrue(): E = default
  def tmFalse(): E = default
  def tmIf(e1: R, e2: R, e3: R): E = default
}
trait Transform[A[-X, Y] <: Term[X, Y]] extends Term[Exp[A], Exp[A]] {
  def tmTrue(): Exp[A] = TmTrue[A, A]()
  def tmFalse(): Exp[A] = TmFalse[A, A]()
  def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
    TmIf[A, A](apply(e1), apply(e2), apply(e3))
}
trait Lifter[-R, E, C] extends Term[R, C => E] {
  def propagate(c: C): Term[R, E]
  def tmTrue(): C => E = propagate(_).tmTrue()
  def tmFalse(): C => E = propagate(_).tmFalse()
  def tmIf(e1: R, e2: R, e3: R): C => E = propagate(_).tmIf(e1, e2, e3)
}
trait Inject[A[-X, Y] <: Term[X, Y], B[-R, _]]
  extends Term[Exp[B], Option[SExp[A, B]]] {
  def tmTrue(): Option[SExp[A, B]] = Some(TmTrue[A, B]())
  def tmFalse(): Option[SExp[A, B]] = Some(TmFalse[A, B]())
  def tmIf(e1: Exp[B], e2: Exp[B], e3: Exp[B]): Option[SExp[A, B]] =
    Some(TmIf[A, B](e1, e2, e3))
}
trait Convert[A[-X, Y] <: Term[X, Y]] {
  def convertBool[B[-R, _]](e: SExp[A, B]): Option[SExp[Term, B]]
}
```

FIGURE 6.3: Traversal templates, lifter, and utilities for type conversion in a language component.

```
package component.bool
...
trait Parse[A[-X, Y] <: Term[X, Y]] extends BaseParser {
  lexical.reserved += ("true", "false", "if", "then", "else")
  lexical.delimiters += ("(", ")")
  private lazy val pTrue = "true" ^^^ TmTrue[A, A]()
  private lazy val pFalse = "false" ^^^ TmFalse[A, A]()
  private lazy val pIf =
    ("if" ~> pE) ~ ("then" ~> pE) ~ ("else" ~> pE) ^^
      { case e1 ~ e2 ~ e3 => TmIf[A, A](e1, e2, e3) }
  lazy val pBoolE: PackratParser[Exp[A]] =
    pTrue ||| pFalse ||| pIf ||| "(" ~> pE <~ ")"
  lazy val pE: PackratParser[Exp[A]] = pBoolE
}
```

FIGURE 6.4: Parser in a language component.

```
package component.bool
...
trait Eval[A[-X, Y] <: Term[X, Y]] extends Term[Exp[A], Exp[A]]
  with Convert[A] {
  val isVal: A[Exp[A], Boolean]
  def tmTrue(): Exp[A] = TmTrue[A, A]()
  def tmFalse(): Exp[A] = TmFalse[A, A]()
  def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
    if (e1(isVal)) {
      val c = convertBool(e1).getOrElse(sys.error("Conversion failed"))
      c(new Term[Exp[A], Exp[A]] {
        def tmTrue(): Exp[A] = e2
        def tmFalse(): Exp[A] = e3
        def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] =
          sys.error("Not a value")
        def apply(e: Exp[A]): Exp[A] = sys.error("impossible")
      })
    }
    else
      TmIf[A, A](apply(e1), e2, e3)
}
trait IsVal[A[-R, _]] extends Query[Exp[A], Boolean] {
  def default: Boolean = false
  override def tmTrue(): Boolean = true
  override def tmFalse(): Boolean = true
}
trait Print[A[-R, _]] extends Term[Exp[A], String] {
  def tmTrue(): String = "true"
  def tmFalse(): String = "false"
  def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): String =
    s"if (${apply(e1)}) then (${apply(e2)}) else (${apply(e3)})"
}
```

FIGURE 6.5: Operations in a language component.

In Figure 6.3, we define traversal templates, lifter, and utilities for type conversion. The `Query` traversal template is for operations that have a fallback value, so that we can only focus on interesting cases and all the other cases are automatically set to the default value. The `Transform` template is for operations that transform ASTs for certain cases. Similarly, only non-trivial cases need to be implemented, and when it encounters other cases it will simply copy the structure. Readers may refer to the original work of Shy framework [71] for more details about traversal templates.

A parser is defined in Figure 6.4. Operations including evaluation and printing are defined in Figure 6.5. Finally, we group all the elements together in a Scala package, namely `component.bool`.

6.2 Composition and Direct Dependencies

Our language components fully support three forms of language composition: language extension, language unification, and extension composition [15]. By definition, we also have language restriction since "language extension subsumes language restriction" [15]. However, their definition is just a weak form that uses a restriction phrase to narrow the syntax. Strong language restriction that actually drops syntax structures is not supported, because we cannot delete a method from the visitor interfaces.

In other words, we can extend a language linearly by adding components to it, and we can also combine independent extensions and languages together. We have a uniform representation of languages and extensions, so that we usually do not need to distinguish them.

For example, suppose we have a package named `component.bool` which contains all structures from Figure 6.2 to 6.5, and another package named `component.utlc` which also has all the structures representing untyped lambda calculus. We can compose them together to build a small language `utlcbool`, as shown in Figure 6.6.

Most of the code glue corresponding structures together using straightforward multiple inheritance. In some structures such as parsers, we need more modifications.

Direct Dependencies and Factories The composition of language components introduces dependencies between them. Dependency relationships can be described as an acyclic graph. For example, Figure 6.7 shows dependencies after composing `utlcbool` and another component `X` to build language `Y`.

We can further divide dependencies into two categories: *direct* and *indirect* dependencies. For the language `Y`, it has indirect dependencies on `bool` and `utlc`, and a direct dependency on `X`. If possible, we only want to consider direct dependencies

```
package component.utlcbool
...
import component.{bool, utlc}

trait Term[-R, E] extends bool.Term[R, E] with utlc.Term[R, E]

trait Query[-R, E] extends Term[R, E]
  with bool.Query[R, E] with utlc.Query[R, E]
trait Transform[A[-X, Y] <: Term[X, Y]] extends Term[Exp[A], Exp[A]]
  with bool.Transform[A] with utlc.Transform[A]
trait Lifter[-R, E, C] extends Term[R, C => E]
  with bool.Lifter[R, E, C] with utlc.Lifter[R, E, C] {
  override def propagate(c: C): Term[R, E]
}
trait Inject[A[-X, Y] <: Term[X, Y], B[-R, _]]
  extends Term[Exp[B], Option[SExp[A, B]]]
  with bool.Inject[A, B] with utlc.Inject[A, B]
trait Convert[A[-X, Y] <: Term[X, Y]] {
  def convertUtlcbool[B[-R, _]](e: SExp[A, B]): Option[SExp[Term, B]]
}
trait ConvertChainBool[A[-X, Y] <: Term[X, Y]]
  extends bool.Convert[A] with Convert[A] {
  def convertBool[B[-X, Y]](e: SExp[A, B]): Option[SExp[bool.Term, B]] = ...
}
trait ConvertChainUtlc[A[-X, Y] <: Term[X, Y]]
  extends utlc.Convert[A] with Convert[A] {
  def convertUtlc[B[-X, Y]](e: SExp[A, B]): Option[SExp[utlc.Term, B]] = ...
}

trait Eval[A[-X, Y] <: Term[X, Y]] extends Term[Exp[A], Exp[A]]
  with bool.Eval[A] with utlc.Eval[A] with ConvertChainBool[A]
trait IsVal[A[-R, _]] extends Query[Exp[A], Boolean]
  with bool.IsVal[A] with utlc.IsVal[A]
trait Print[A[-R, _]] extends Term[Exp[A], String]
  with bool.Print[A] with utlc.Print[A]

trait Parse[A[-X, Y] <: Term[X, Y]] extends bool.Parse[A] with utlc.Parse[A] {
  ...
}
```

FIGURE 6.6: Composition of two language components.

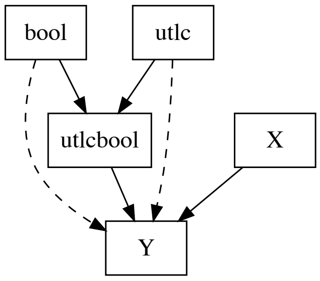


FIGURE 6.7: Direct (solid arrow) and indirect (dashed arrow) dependencies.

```
package component.bool
...
trait Factory {
  type TmTrue[A[-X, Y] <: Term[X, Y], B[-R, _]] = component.bool.TmTrue[A, B]
  val TmTrue = component.bool.TmTrue
  type TmFalse[A[-X, Y] <: Term[X, Y], B[-R, _]] = component.bool.TmFalse[A, B]
  val TmFalse = component.bool.TmFalse
  type TmIf[A[-X, Y] <: Term[X, Y], B[-R, _]] = component.bool.TmIf[A, B]
  val TmIf = component.bool.TmIf
}
object Factory extends Factory
```

FIGURE 6.8: A wrapper for factory classes defined in Figure 6.2.

in compositions, because they behave like abstract interfaces hiding the details of indirect ones.

Using the pattern demonstrated in Figure 6.6, all the structures can be composed considering only direct dependencies except factories. Because Scala does not have a mechanism to export imported classes, it is not straightforward to pack factory classes in a composition.

Our approach is to wrap their references in traits. Figure 6.8 shows a wrapper trait containing factory classes defined in Figure 6.2, with an object for instantiation. The reason we do not put the classes in the trait but use type synonyms and fields to refer them, is to avoid creation of new type identities by Scala's path-dependent types. Otherwise, `component.bool.Factory.TmTrue` would be a totally different type with `component.utlcbool.Factory.TmTrue`, thus the ASTs would not be reuseable.

In `utlcbool`, factories from `bool` and `utlc` can be composed as shown below. All classes are inherited in the trait `Factory` and they can be referred under the object `Factory`.

```
package component.utlcbool
...
trait Factory extends bool.Factory with utlc.Factory
object Factory extends Factory
```

Visitor interface:

```
@Lang("⟨Name⟩")
trait ⟨Interface⟩[-⟨R⟩, ⟨E⟩] extends ⟨Parent⟩[⟨R⟩, ⟨E⟩] {
  def ⟨method⟩(⟨x⟩: ⟨T⟩): E
}
```

Meta-variables:

`⟨Name⟩`	The component name
`⟨Interface⟩`	The interface name
`⟨Parent⟩`	Name iterating parent interfaces.
`⟨OtherParents⟩`	Name iterating interfaces of the other parents for a `⟨Parent⟩`.
`⟨PComponent⟩`	The component name of a `⟨Parent⟩`
`⟨method⟩`	Name iterating methods, the first letter is in lower case
`⟨Method⟩`	`⟨method⟩` with the first letter capitalized
`⟨R⟩` and `⟨E⟩`	Type parameters
`⟨x⟩` and `⟨T⟩`	Names iterating parameters and their types of a method

Transformation for every pair of parameter and type:

$[\![\langle\texttt{T}\rangle]\!]_A$	Replace `⟨R⟩` in `⟨T⟩` by `Exp[A]`
$[\![\langle\texttt{T}\rangle]\!]_B$	Replace `⟨R⟩` in `⟨T⟩` by `Exp[B]`
$(\!|\langle\texttt{x}\rangle|\!)$	When `⟨R⟩` is replaced, change the corresponding expression `e` to `apply(e)`

FIGURE 6.9: Meta-variables and notations for code generation (simplified).

Now language components can be used only considering direct dependencies, and consistency of types is preserved.

6.3 Eliminating Boilerplate by Metaprogramming

While the operations and parsers in a language component are customized by the user, many other structures have a fixed correspondence with the visitor interface. Specifically, the factory classes in Figure 6.2, their wrapper in Figure 6.8, and the traversal templates and lifter in Figure 6.3 are derived from the visitor interface in a straightforward way. The corresponding glue code in compositions, such as line 7 to 28 in Figure 6.6, are also closely related with the inheritance of interfaces.

The visitor interface itself is enough to represent the abstract syntax in a language component. That allows us to generate aforementioned boilerplate code automatically by metaprogramming. In particular, we employ Scala's macro annotation [9] and Scalameta toolkit [55] to annotate visitor interfaces, and then generate companion objects containing those boilerplate code. That is the metaprogramming library part of our framework Gems.

Figure 6.9 shows meta-variables and notations for demonstration. Meta-variables are in angle brackets, and the overline means multiple appearances. Figure 6.10 presents template of generated companion object, which contains boilerplate code of factories, traversal templates, and lifter. For readability, the demonstration is simplified by using single-sorted abstract syntax, although our actual implementation is powerful enough to handle multiple sorts.

The visitor interface may inherit from other parent interfaces, representing direct dependencies between components. As discussed in Section 6.2, we only need to consider these direct dependencies for composing structures. Therefore, our code generation requires only local and syntactical information of interface inheritance, rather than global and semantical analysis deeply into existing components.

In client code, the user only needs to annotate macro `@Lang` on the visitor interface with the name of component, then all generated structures are accessible via the companion object.

For example, if we annotate a `Term` interface as below, an operation `IsVal` could use the query template by extending `Term.Query`.

```
@Lang("name")
trait Term[-R, E] extends ... {...}
trait IsVal[A[-R, _]] extends Term.Query[Exp[A], Boolean] with ... {...}
```

If we annotate `@Lang("bool")` to the `Term` interface in Figure 6.2, the metaprogramming library of Gems will generate a companion object in Figure 6.11. As we can see, the meta-variables are instantiated by corresponding names, and then we have factory classes, traversal templates, lifter, and conversion utilities for free.

Code generation is not limited to the boilerplate structures we listed, and the user could adjust the annotation processor to incorporate more structures as needed.

```
object ⟨Interface⟩ {
  case class ⟨Method⟩[A[-X, Y] <: ⟨Interface⟩[X, Y], B[-R, _]](⟨x⟩: ⟦⟨T⟩⟧)
    extends SExp[A, B] {
    def apply[E](vis: A[Exp[B], E]): E = vis.⟨method⟩(⟨x⟩)
  }

  trait Factory extends ⟨Parent⟩.Factory {
    type ⟨Method⟩[A[-X, Y] <: ⟨Interface⟩[X, Y], B[-R, _]] =
      ⟨Interface⟩.⟨Method⟩[A, B]
    val ⟨Method⟩ = ⟨Interface⟩.⟨Method⟩
  }
  object Factory extends Factory

  trait QueryThis[-⟨R⟩, E] extends ⟨Interface⟩[⟨R⟩, E] with Default[E] {
    def ⟨method⟩(⟨x⟩: ⟨T⟩): E = default
  }
  trait Query[-⟨R⟩, E] extends ⟨Interface⟩[⟨R⟩, E]
    with ⟨Parent⟩.Query[⟨R⟩, E] with QueryThis[⟨R⟩, E]

  trait Transform[A[-X, Y] <: ⟨Interface⟩[X, Y]]
    extends ⟨Interface⟩[Exp[A], Exp[A]] with ⟨Parent⟩.Transform[A] {
    def ⟨method⟩(⟨x⟩: ⟦⟨T⟩⟧_A): Exp[A] = ⟨Method⟩[A, A](⦇⟨x⟩⦈)
  }

  trait Lifter[-⟨R⟩, E, C] extends ⟨Interface⟩[⟨R⟩, C => E]
    with ⟨Parent⟩.Lifter[⟨R⟩, E, C] {
    def propagate(c: C): ⟨Interface⟩[⟨R⟩, E]
    def ⟨method⟩(⟨x⟩: ⟨T⟩): C => E = propagate(_).⟨method⟩(⟨x⟩)
  }

  trait Inject[A[-X, Y] <: ⟨Interface⟩[X, Y], B[-R, _]]
    extends ⟨Interface⟩[Exp[B], Option[SExp[A, B]]] {
    def ⟨method⟩(⟨x⟩: ⟦⟨T⟩⟧_B): Option[SExp[A, B]] = Some(⟨Method⟩[A, B](⟨x⟩))
  }

  trait Convert[A[-X, Y] <: ⟨Interface⟩[X, Y]] {
    def convert⟨Name⟩[B[-R, _]](e: SExp[A, B]): Option[SExp[⟨Interface⟩, B]]
  }

  trait ConvertChain⟨PComponent⟩[A[-X, Y] <: ⟨Interface⟩[X, Y]]
    extends ⟨Parent⟩.Convert[A] with Convert[A] {
    def convertBool[B[-X, Y]](e: SExp[A, B]): Option[SExp[⟨Parent⟩, B]] = {
      val t = new ⟨Interface⟩[Exp[B], Option[SExp[⟨Parent⟩, B]]]
        with ⟨Parent⟩.Inject[⟨Parent⟩, B]
        with QueryThis[Exp[B], Option[SExp[⟨Parent⟩, B]]]
        with ⟨OtherParents⟩.Query[Exp[B], Option[SExp[⟨Parent⟩, B]]] {
        def default: Option[SExp[⟨Parent⟩, B]] = None
        def apply(e: Exp[B]): Option[SExp[⟨Parent⟩, B]] = sys.error("Impossible")
      }
      convert⟨Name⟩[B](e).flatMap(_ (t))
    }
  }
}
```

FIGURE 6.10: Companion object generated from the interface in Figure 6.9 (simplified).

```
object Term {
  case class TmTrue[A[-X, Y] <: Term[X, Y], B[-R, _]]() extends SExp[A, B] {
    def apply[E](alg: A[Exp[B], E]): E = alg.tmTrue()
  }
  case class TmFalse[A[-X, Y] <: Term[X, Y], B[-R, _]]() extends SExp[A, B] {
    def apply[E](alg: A[Exp[B], E]): E = alg.tmFalse()
  }
  case class TmIf[A[-X, Y] <: Term[X, Y], B[-R, _]](e1: Exp[B], e2: Exp[B], e3:
    Exp[B]) extends SExp[A, B] {
    def apply[E](alg: A[Exp[B], E]): E = alg.tmIf(e1, e2, e3)
  }
  trait QueryThis[-R, E] extends Term[R, E] with Default[E] {
    def tmTrue(): E = default
    def tmFalse(): E = default
    def tmIf(e1: R, e2: R, e3: R): E = default
  }
  trait Query[-R, E] extends Term[R, E] with QueryThis[R, E]
  trait Factory {
    type TmTrue[A[-X, Y] <: Term[X, Y], B[-R, _]] = Term.TmTrue[A, B]
    val TmTrue = Term.TmTrue
    type TmFalse[A[-X, Y] <: Term[X, Y], B[-R, _]] = Term.TmFalse[A, B]
    val TmFalse = Term.TmFalse
    type TmIf[A[-X, Y] <: Term[X, Y], B[-R, _]] = Term.TmIf[A, B]
    val TmIf = Term.TmIf
  }
  object Factory extends Factory
  trait Transform[A[-X, Y] <: Term[X, Y]] extends Term[Exp[A], Exp[A]] {
    def tmTrue(): Exp[A] = TmTrue[A, A[-?, ?]]()
    def tmFalse(): Exp[A] = TmFalse[A, A[-?, ?]]()
    def tmIf(e1: Exp[A], e2: Exp[A], e3: Exp[A]): Exp[A] = TmIf[A, A[-?, ?]](
    apply(e1), apply(e2), apply(e3))
  }
  trait Lifter[-R, E, C] extends Term[R, C => E] {
    def propagate(c: C): Term[R, E]
    def tmTrue(): C => E = propagate(_).tmTrue()
    def tmFalse(): C => E = propagate(_).tmFalse()
    def tmIf(e1: R, e2: R, e3: R): C => E = propagate(_).tmIf(e1, e2, e3)
  }
  trait Inject[A[-X, Y] <: Term[X, Y], B[-R, _]] extends Term[Exp[B], Option[
    SExp[A, B]]] {
    def tmTrue(): Option[SExp[A, B]] = Some(TmTrue[A, B]())
    def tmFalse(): Option[SExp[A, B]] = Some(TmFalse[A, B]())
    def tmIf(e1: Exp[B], e2: Exp[B], e3: Exp[B]): Option[SExp[A, B]] = Some(TmIf
    [A, B](e1, e2, e3))
  }
  trait Convert[A[-X, Y] <: Term[X, Y]] {
    def convertBool[B[-R, _]](e: SExp[A, B]): Option[SExp[Term, B]]
  }
}
```

FIGURE 6.11: Companion object generated from the `Term` interface in Figure 6.2.

Chapter 7

Case Study

This chapter presents a case study to demonstrate the utility of our language modularization framework Gems. We implement interpreters for the first 18 languages[1] from the *Types and Programming Languages* (TAPL) book [46]. Common language structures are abstracted from those languages as reusable components to eliminate duplications. The interpreters are built as full pipelines containing parsers, typing, evaluation, printing, and other auxiliary utilities. We compare our modular implementation with an ordinary non-modular one available online, which is also written in Scala using the same parsing library. Source lines of code (SLOC) and execution time are measured for both implementations. The results suggest that our modular implementation saves 59.9% of code, and it has 118.2% and 143.6% slower execution time for parsing and evaluation, respectively.

7.1 Overview

The TAPL book introduces several languages from simple to complex, by gradually adding new features. The idea of using TAPL in case study is borrowed from EVF [72]. However, their case study compares implementations written in two different languages (Java and OCaml), and thus they only compare the size of code but not execution time. Furthermore, their interpreters do not include parsers.

The languages in TAPL are suitable for our case study for mainly three reasons. Firstly, they capture many of the language features required in realistic programming languages, such as functions, records and polymorphism. Secondly, the evolution of languages in the book reveals advantages of language modularization, because we can reuse common features from those languages. Thirdly, the case study poses several challenges to modularity, including multi-sorted abstract syntax, dependent operations and operations with contexts.

[1]There are some more languages in the book, but they are either not ported by the implementation we compare with, or do not introduce new syntax structures.

7.1.1 Languages

Table 7.1 gives brief description of all the languages in our case study. It starts from a simple arithmetic expression language `arith` and ends with an extended version of System F_ω named `fullomega`.

Multiple Sorts Those languages have different number of sorts in the abstract syntax. The first three languages have only one sort: *terms* (expressions). Most other languages have two sorts *terms* and *types*, except the last one `fullomega` which also has *kinds*. The abstract syntax of those multi-sorted languages are defined using the approach discussed in Section 4.5. As a result, every sort in the syntax of a language is extensible, and subtyping relationships are properly preserved.

Another important aspect of those languages is typing. The first three single-sorted languages are untyped. Some of the typed languages have subtyping (marked by "S"), while some other are simply typed (marked by "T"). Two languages have type reconstruction (marked by "R"), and types can be inferred.

Many languages are based on simply typed lambda calculus (STLC), which consists of lambda abstraction, variables, lambda application and types. In addition, some structures such as boolean expressions and records are shared among several languages. We benefit greatly from such sharing using Gems. Moreover, since different sorts are independently modularized, we often have flexibility to specify what types and what type checking procedure we need, while reusing structures of terms.

7.1.2 Interpreter Structure

For every language we implement an interpreter from parsing to evaluation with several other facilities. Specifically, we implement the *abstract syntax*, *parser*, *printing* operation and *evaluation* operation for all languages. The evaluation operation depends on two auxiliary operations: *substitution* which replaces a variable by a term, and *is-value* which tests whether a term is a value and thus cannot be reduced anymore.

We also implement a *typing* operation for those typed languages, and it varies according to the type system of each language. For simply typed languages, a *type-equivalence* operation is introduced to check if two types are equal. A *subtype-of* operation is used in several languages that have subtyping relationships. It depends on two operations: *meet* for finding a closest common subtype and *join* for finding a closest common supertype. *Meet* and *join* are mutually dependent on each other and they both depend on the *subtype-of* operation. Moreover, some languages need a *type-substitution* operation to substitute type variables. The last language `fullomega` requires *kinding* and *kind-equivalence* operations in addition.

Name	**# of sorts**	**Typing**	**Description**
arith	1		Expressions of booleans and natural numbers
untyped	1		Untyped lambda calculus
fulluntyped	1		`untyped` plus `arith` and extensions
tyarith	2	T	`arith` with type checker
simplebool	2	T	STLC with boolean expressions
fullsimple	2	T	STLC with extensions (variants, records, etc.)
bot	2	S	STLC with top and bottom types
fullerror	2	S	`bot` with bottom value as errors
rcdsubbot	2	S	`bot` with record subtyping
fullsub	2	S	Subtyping on extended STLC
fullref	2	S	`fullsub` with mutable references
equirec	2	T	STLC with equi-recursive types
fullequirec	2	T	`equirec` with extensions
fullisorec	2	T	Use iso-recursive types in `fullequirec`
recon	2	R	Type reconstruction on STLC and `tyarith`
fullrecon	2	R	`recon` with extensions
fullpoly	2	T	System F with extensions
fullomega	3	T	System F_ω with extensions

TABLE 7.1: Languages in the case study.

T = Typing. S = Subtyping. R = Type reconstruction.

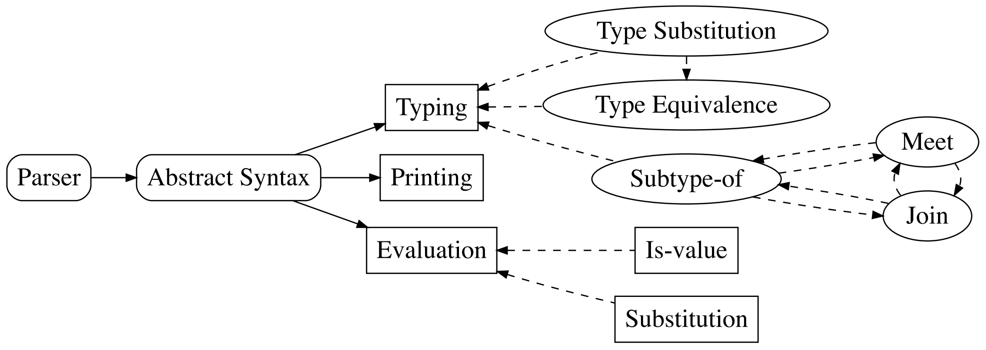


FIGURE 7.1: Interpreter structure with major operations.

Boxes = operations on terms. Circles = operations on types.
Dashed arrows = dependencies.

Operation Dependencies As we described, many operations collaborate with and depend on other operations. Figure 7.1 illustrates the structure of interpreters and major operations in our implementation. Boxes (with sharp corners) are operations on the term level, and circles are operations on the type level. Dependencies between operations are represented by dashed arrows. The technique we discussed in Section 4.3 is used to solve the challenge of having dependent operations while retaining extensibility for each one.

Operations with Contexts Another interesting issue is that the type of an operation may change in different languages, because it requires more contexts. For example, the evaluation operation in language `fullref` needs an extra context for storing references. This issue also happens on the type level. It is straightforward to decide whether two simple types are equivalent, such as the boolean and unit type. However, if more complex types such as recursive types are introduced, the type equivalence operation needs contexts. In our implementation, we use context propagation (see Section 4.6) for adding contexts when reusing old operations.

7.1.3 Components and Dependencies

Using our framework Gems, we extract common features from those languages as reusable components. The components are listed in Table 7.2 with corresponding usage count in all languages. The count numbers illustrate great reuse in the whole implementation. For example, the `nat` component is used in 12 languages and that means 11 times of duplication are eliminated.

There are 17 components which contain language structures, and one special component named `extension` is just a bundle of some others for more concise composition. Figure 7.2 demonstrates dependencies of all the 18 languages and 18 components in the case study. White boxes represent components and grey boxes represent languages in the figure.

Name	**# of usage**	**Description**
nat	12	Natural numbers
bool	14	Boolean literals and if-then-else
bottom	3	The bottom type
record	9	Records (labeled products) and projections
varapp	16	Variables and function applications
let	9	"let..in.." structures
typedrecord	8	`record` with record types
pack	2	Pack and unpack structures for existential types
typednat	10	`nat` with a type of numbers
typed	14	Simply typed lambda calculus (STLC) skeleton
ref	2	Structures of references
typedbool	12	`bool` with a boolean type
unit	7	The unit value and type
variant	4	Variants (labeled sums) and case analysis
floatstring	8	Floating numbers and strings
top	5	The top type
rectype	3	Recursive types
extension	7	`tyarith`, `floatstring`, `let`, `typedrecord` and `unit`

TABLE 7.2: Components in the case study.

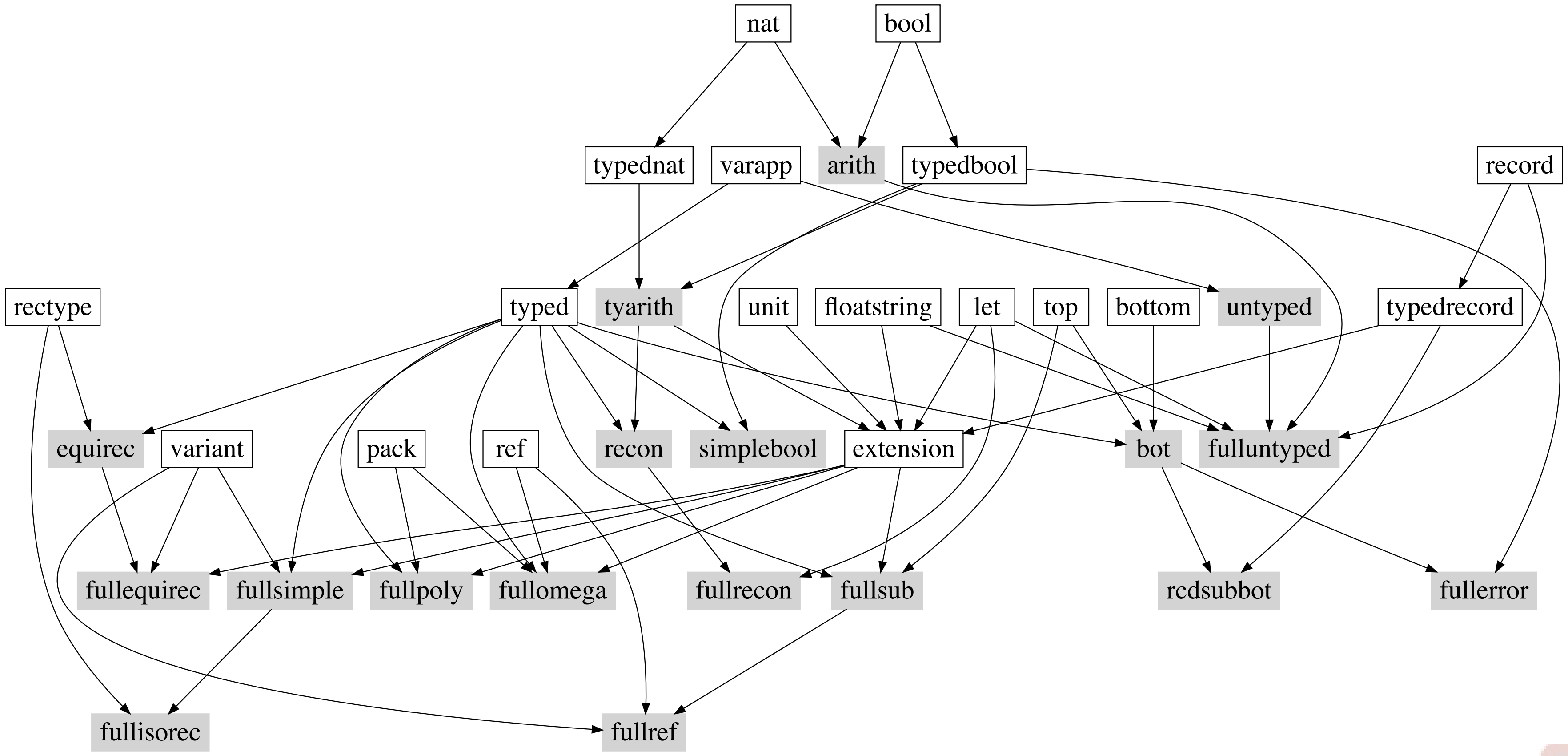


FIGURE 7.2: Dependencies between all languages and components in the case study.

White boxes = components. Grey boxes = languages.

7.2 Comparison

We compare our implementation using Gems (named `Modular`) with a non-modular implementation available online[2] (named `NonMod`). `NonMod` is suitable for comparison because it also implements TAPL interpreters in Scala using the same parsing library. The languages in `NonMod` are built independently from each other, thus they do not reuse any code even when many languages are overlapped.

The comparison consists of two aspects. Firstly, we want to discover the extent of reuse with language modularization. A direct measurement for that is source lines of code (SLOC). Although the benefit of reuse is not limited to code reduction, SLOC straightforwardly reflects how much effort users could save in terms of programming. Secondly, we are interested to assess the performance penalty caused by language modularization. Intuitively, visitors may be slower than plain functions because they have more intermediate steps when performing an operation. Therefore, we compare execution time of the interpreters in the two implementations with same inputs.

7.2.1 SLOC Comparison

In the SLOC comparison, all blank lines and comments are excluded, and the code of both implementations is formatted to guarantee that the length of each line does not exceed 120 characters. Furthermore, `NonMod` has some extra functionalities than ours, such as interpretation for a whole file. Such code is removed before comparison.

Table 7.3 shows comparison results. We compare our implementation `Modular` with `NonMod` for every language. The percentage of difference is calculated by:

$$(\texttt{Modular} - \texttt{NonMod})/\texttt{NonMod} * 100\%$$

To make the comparison more comprehensive, we calculate two set of numbers. The column `Modular`^{a} contains the lines of code for only the languages themselves, excluding all imported language components. The column `Modular`^{b} shows the lines of code for standalone programs of each language. That means we count lines of code of all required components in the numbers.

We can see that the code of `Modular` is considerably shorter because components are reused among languages. The total numbers including all of the code illustrate that `Modular` reduces 59.9% lines of code overall.

Moreover, if we suppose the components are available in an existing library, we only need to glue them and add few extra structures to build a language. In that case, we have even better code reduction rate 76.4%. For example, the language `fullsimple` is

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

Language	NonMod	Modular[a]	(+/-)%	Modular[b]	(+/-)%
arith	124	30	-75.8%	263	+112.1%
untyped	145	64	-55.9%	212	+46.2%
fulluntyped	310	45	-85.5%	576	+85.8%
tyarith	180	56	-68.9%	300	+66.7%
simplebool	244	69	-71.7%	359	+47.1%
fullsimple	716	65	-90.9%	571	-20.3%
bot	229	82	-64.2%	295	+28.8%
fullerror	439	131	-70.2%	500	+13.9%
rcdsubbot	312	78	-75.0%	466	+49.4%
fullsub	731	107	-85.4%	526	-28.0%
fullref	996	208	-79.1%	966	-3.0%
equirec	279	102	-63.4%	346	+24.0%
fullequirec	731	137	-81.3%	753	+3.0%
fullisorec	756	117	-84.5%	716	-5.3%
recon	391	145	-62.9%	571	+46.0%
fullrecon	413	109	-73.6%	726	+75.8%
fullpoly	719	143	-80.1%	572	-20.4%
fullomega	931	352	-62.2%	885	-4.9%
total	8646	2040	-76.4%	3466[c]	-59.9%

[a] Lines of code for only the language, excluding all imported components.
[b] Lines of code for a standalone program, counting all imported components.
[c] Lines of code of the whole project. Each component is counted only once.

TABLE 7.3: Source lines of code (SLOC) comparison results.

a fortunate one, in which we only glue components and save 90.9% of code. It suggests that if we build a relatively rich library of components representing common language features, we can quickly develop a new language using few lines of code. The more powerful the library is, the more easily users create a new language.

7.2.2 Execution Time Comparison

We test the performance of parsing and evaluation operation of every interpreter in both implementations. Random expressions are generated as test cases according to the features of each language and stored in files. For both implementations, parsing time of 1000 random expressions and evaluation time of 10000 random expressions are measured in milliseconds. The size of parsing test files varies from 41 KB to 212 KB (each file corresponds to a language), and the size of evaluation test files varies from 2.2 MB to 10 MB.

We use ScalaMeter [56] benchmarking framework for testing. Default warm-ups provided by the framework is applied and results are measured by the average of 10 runs. Time of reading inputs is excluded. Tests are performed using Scala 2.12.3, Java 1.8.0_131, on a MacBook Pro with 2.9 GHz Intel Core i5 processor (6267U) and 8 GB RAM.

Language	Parsing			Evaluation		
	`Modular`	`NonMod`	(+/-)%	`Modular`	`NonMod`	(+/-)%
arith	372.5	196.4	89.7%	20.5	6.9	197.1%
untyped	377.4	203.5	85.5%	60.0	34.6	73.4%
fulluntyped	1615.8	662.2	144.0%	216.0	69.6	210.3%
tyarith	376.3	201.9	86.4%	23.2	7.3	217.8%
simplebool	1580.9	786.6	101.0%	283.5	110.3	157.0%
fullsimple	2311.0	1062.3	117.5%	189.2	78.6	140.7%
bot	763.1	490.7	55.5%	73.1	35.0	108.9%
fullerror	1753.5	825.3	112.5%	70.4	38.9	81.0%
rcdsubbot	1102.5	605.6	82.1%	133.7	49.0	172.9%
fullsub	1833.1	794.9	130.6%	212.7	73.7	188.6%
fullref	2390.6	972.6	145.8%	292.6	119.3	145.3%
equirec	703.0	409.1	71.8%	85.4	60.3	41.6%
fullequirec	2453.3	1048.8	133.9%	195.3	75.3	159.4%
fullisorec	2395.4	1120.8	113.7%	174.4	69.4	151.3%
recon	1015.3	523.3	94.0%	174.8	59.7	192.8%
fullrecon	1260.9	557.3	126.3%	134.6	45.8	193.9%
fullpoly	1991.6	823.6	141.8%	213.8	97.2	120.0%
fullomega	2332.9	920.9	153.3%	314.1	146.0	115.1%
total	26629.1	12205.8	118.2%	2867.3	1176.9	143.6%

TABLE 7.4: Execution time comparison results. (in milliseconds)

Table 7.4 shows comparison results. The percentage numbers are calculated using the same formula as in the SLOC comparison.

In the comparison of parsing time, `Modular` is slower in every case and the overall result suggests that it spends 118.2% more time than `NonMod`. The use of left recursion and creation of intermediate parsers are probably the reasons.

The results of evaluation time comparison demonstrate that `Modular` has 143.6% slower performance overall. Comparing with normal functions, visitors have more steps to perform an operation, since it relies on the *accept* method to select the corresponding *visit* method.

Furthermore, visitors are harder to be optimized in Scala than normal functions. For example, Scala has tail-call optimizations (TCO) to accelerate recursive functions. The following code fragment shows a function `isVal` in language `fullsimple` of the `NonMod` implementation, where a tail-call optimization is possible.

```
def isVal(ctx: Context, t: Term): Boolean = t match {
  ...
  case TmTag(_, t1, _) => isVal(ctx, t1)
  ...
}
```

It is usually difficult to apply such optimizations using visitors because of the indirect steps. Especially in Scala, TCO is limited to a direct recursive call on a function itself as its last operation. Thus, an operation `isVal` implemented by visitors will not be optimized.

Although it depends on the situation, we believe the slower execution time is usually a worth trade for modularity, because it benefits both maintenance and future extensions. More analysis on performance and optimization are left for future work.

Chapter 8

Related Work and Conclusion

8.1 Related Work

8.1.1 Modular Parsing

Syntactically Extensible Parsing There have been several tools to modularize syntax and corresponding parsers using extensible parser generators [45, 25, 58, 62, 24, 69]. They support composition of grammars and productions. Some of them such as ANTLR [45], Rats! [25] enable users to override rules in extensions.

Those parser generators stay on the syntactical level of modularity. The result parsers are generated by analyzing the composed grammar which often uses textual representations. Even if the grammar is only modified a bit, it requires a full compilation for building corresponding parsers. For example, NOA [24] first collects all syntax definitions by Java annotations, and then generates grammars for ANTLR parsing. Modular type-checking of parsers is often absent in those tools as well. The work [62] presents grammar and production compositions in a static typed manner. They build a Haskell library containing utilities for specifying syntax, and grammars are guaranteed to preserve proper types. Generation of parsers is still performed after composition. An advantage of using parser generators is that they are able to perform useful analysis such as ambiguity detection on grammars, while it is hard to achieve this without access to grammars.

Combination of Parsers Instead of working on grammars, another way to achieve extensibility is to directly combine parsers. The techniques of parse table composition [5, 59] allow users to compose DFA and NFA parse tables. Therefore, parsers represented by tables can be separately compiled. Since users do not want to construct parse tables by hand, this approach requires synchronization with grammar definitions during evolution of the syntax, but the existing grammars do not need to be touched. One potential issue is that it may not be easy to type-check the tables and companion code for assembling ASTs, and we found little discussion about that in those works.

Parser combinators are adopted by us for modular parsing in Chapter 5. Parser combinators can be traced to the work of Burge [8] in 1975, and they have been studied in a lot of other work [64, 28, 18, 29] in the following years. Usually, parsers are represented by functions using parser combinator libraries. That makes it easier to write them by hand, which is the normal way in practice, and closer to the host language because the generation step is avoided. The Parsec [39] library in Haskell is a very famous one. However, it uses an algorithm of simple recursive descent parsing and manual backtracking. That makes it inconvenient in a modular setting. Advanced parser combinator techniques [19, 21] have been proposed to improve issues of simple recursive descent parsing. We use Packrat parsing [19, 68] because it is suitable for most cases in modular parsing and is accessible in Scala.

8.1.2 Datatype Modularization and Visitors

Extensible Datatypes in Functional Programming The expression problem [66] poses a challenge to achieve semantic modularity when extending datatypes. In functional programming, a famous solution is Datatypes à la Carte (DTC) [60], with some following variants [3, 2, 51]. The different "shapes" of recursive datatypes, which are different variants in a datatype, are represented by functors and composed freely. DTC has a projection mechanism to inspect the outermost structure of a datatype instance, and convert its type to a more concrete one. That is very similar to the type conversion technique we used in Section 4.7 to perform ad-hoc case analysis. A difference is that they do not have real transitive subtyping relationships for datatypes while we have. Another approach is final tagless [10], which essentially uses the same encoding as Object Algebras. It uses type classes in Haskell to represent interfaces carrying abstract methods.

Visitors There have been several works to make the visitor design pattern extensible in a type-safe way and thus give a solution to the expression problem [61, 44, 49]. Those works use different features of type system and programming language. The solutions of Torgersen [61] mainly depend on generic techniques including wildcards and F-bounds. The two solutions proposed by Odersky and Zenger [44] use virtual types and nested class scopes. The solution of modular external visitors [49] needs higher-order type parameters and type variance. We take this solution because it is the most straightforward (and brute-force) one, and has great theoretical support. The datatype instances are encoded by recursive polymorphic function types directly, and subtyping relationships of those types are preserved. Currently in Scala we need auxiliary traits such as the `Exp` to "tie the recursive knot" for the types. However, in a language with stronger support of subtyping with recursive types and type alias, the encoding would be quite elegant to use. The work of EVF [72] is a visitor framework in Java, that relies on metaprogramming to generate AST definitions for each extension. It requires few language features and allows operations to be fully reused. The ASTs are not reusable though. We borrow its idea of abstracting recursive calls and thus eliminate the need of self type annotations.

Scala's case classes [13] provide an encoding of algebraic datatypes and allow the extensibility of adding new constructors. However, case classes which are not `sealed` do not guarantee exhaustiveness of pattern matching and cannot be implemented using traditional OO structures in a type safe manner.

Object Algebras Object Algebras [50] are a lightweight design pattern that solves the expression problem. The technique only requires simple generics and (multiple) inheritance, which are ordinary features in object-oriented programming. There have been several works towards applications [47, 24, 4] and better usage [52, 71] of Object Algebras. Especially, there were some attempts to modularize language syntax and semantics [24, 30]. The Shy framework [71] introduces several traversal templates for Object Algebras to reduce boilerplate code. The work of implicit context propagation [30] presents a solution to add contexts when reusing existing operations in Object Algebras interpreters. As EVF [72] and our work demonstrated, such techniques can also be applied on external visitors. Object Algebras are close to Church encodings and internal visitors. They combine results on recursive positions rather than perform operations on subtrees explicitly. That makes it elegant to write simple folds but hard to express dependencies and let one operation cooperate with others. Moreover, the lack of concrete datatype representation is inconvenient and even inefficient in some cases.

8.1.3 Language Workbenches

Language workbenches [20] are tools that facilitate implementation of (domain-specific) languages. They provide utilities to define, reuse and compose languages, together with auxiliary functionalities such as IDE support, testing and debugging. Therefore, the effort of creating new languages is reduced a lot and language-oriented programming paradigm [67, 11] is advocated. Language workbenches may support many forms of composition, including (linear) language extensions and language unification [15]. Examples of language workbenches include Ensō [14], MPS [43], MetaEdit+ [37], Spoofax [36] and Xtext [17]. Many of them have successful applications in practice. However most of the language workbenches use syntactic modularization techniques to achieve extensibility. For example, In Spoofax [36], the syntax of languages are defined using SDF3 [63], and transformations are defined using Stratego [6]. Then the implementation is generated from those definitions.

Current language workbenches often rely on metaprogramming and code generation to reuse components. The textual input written in meta-languages which describes aspects of the working languages blocks separate compilation and modular static type-checking. Some tools try to strengthen the meta-languages or apply more analysis on the meta-language level to ensure the safety of components. However, users are restricted to work on that level, and the type system is still separate from the language they are targeting. An example is ableC [34], which is a framework to extend the language C. It uses two analysis on the meta-language level to guard the safety of syntax and semantics, and guarantees the safety of compositions.

8.2 Conclusion

In the thesis we present our work of a language modularization framework Gems. We use only pure visitors and achieve semantic modularity in the full pipeline of a language. The approach is quite lightweight overall. For ASTs and semantics, we adopt modular external visitors [49] and make several enhancements. The modular parsing part describes our recent work [70]. We build a simple metaprogramming library that generates boilerplate structures to ease the development. A TAPL case study is conducted to investigate the utility.

The heavy effort of creating new languages and maintain existing languages motivates people to modularize and reuse languages, and that is really desired. When C++ programmers want to use new features they may need to pass arguments such as `-std=c++17` to the compiler. Haskell programmers often put flags on the top of a file to turn on certain language features. We believe that, in many cases, the semantic modularization of language implementations is better than applying generation tools on fragile textual representation of languages. Especially, the static type-safety could detect errors earlier and save a lot of time. The overlap parts among languages can be extracted and reused, thus the total workload is reduced a lot. The results of our case study support that.

We must distinguish those modularization techniques that *employ* metaprogramming and those *rely on* metaprogramming. Techniques in the former category use metaprogramming to generate boring code. Without metaprogramming, the code would be tedious but the logic and extensibility are not affected. Techniques in the latter category really depend on metaprogramming to achieve modularity by copying some code. Without metaprogramming, the extensibility would be lost and programmers would have to repeat logic of existing works. For instance, EVF [72] relies on metaprogramming to avoid repeating existing data variants when extending new ones. Fortunately, we are in the former category, that is we could abandon metaprogramming but still have semantic modularity.

In terms of practical programming, visitors indeed make the code more complex comparing with plain functions, if we do not consider reuse of code. The comparatively weak type inference mechanism of Scala also forces us to add some redundant type annotations. However, it is still acceptable to implement by hand in such a way. Our approach could also be used as an internal representation of a language workbench. Especially considering some language workbenches are implemented using dynamic languages.

To conclude, our work is an attempt towards semantically modularization of languages in practice. We believe this could be a good alternative to the current syntactical approaches based on code generation.

8.3 Future Work

We propose the following directions for future work:

- Analyse the performance of parsing and other operation executions in more detail, and study optimizations to improve the performance. We observed the implementation using visitors have more intermediate steps, since it relies on the *accept* method to select the corresponding *visit* method. Comparing with plain functions, those intermediate function calls slow down the performance of every operation application, including every recursive call, and eliminate the chance of applying tail-call optimizations. A potential direction of improving this issue is *generative programming* or *staging* [33], that dynamically generates and specializes the code. LMS [48] is a library in Scala for that and might be helpful.

- There are still some boilerplate code in component compositions. Especially when composing parsers and operations, the related structures from parents need to be specified repeatedly. For example, we may have the following code:

  ```
  trait Op1[..] extends Interface[..] with Parent1.Op1[..] with Parent2.Op1[..]
      with Parent3.Op1[..] with ..
  trait Op2[..] extends Interface[..] with Parent1.Op2[..] with Parent2.Op2[..]
      with Parent3.Op2[..] with ..
  ```

  It would be better to minimize such glue code of repeating parent structures. This issue seems to be related to family polymorphism [16], because a language contains multiple classes as a family and we are maintaining the relationships between the corresponding classes and families. Thus the techniques of family polymorphism might help. Moreover, a simple workaround is to apply metaprogramming to generate such code, but it is probably too ad-hoc.

- Develop a simple language workbench using Gems as the core for modularization, and study the applicability of it. The workbench may allow users to directly develop in Scala using the patterns we introduced, or let users specify the syntax and (parsing, typing, rewriting, etc.) rules to generate initial implementations of visitors, parsers, and other utilities. Composition of languages could be automated, and users may import components from the library to their languages, or share developed components with others.

- A more ambitious direction is to improve the support of visitors in programming languages. If we had native support of extensible datatypes and visitors, the code would be elegant and concise. The key point is to properly handle two recursions: the recursion of datatypes and recursion of operations (visitors). The recursive references should be bound dynamically to allow future extensions. An imaginary syntax is shown below.

```
data Exp {
  case Lit(x: Int)
  case Add(e1: this, e2: this)
}
visitor Eval extends Exp {
  case Lit(x: Int) => x
  case Add(e1: Exp.this, e2: Exp.this) => this(e1) + this(e2)
}
data Exp2 extends Exp {
  case Sub(e1: this, e2: this)
}
visitor Eval2 extends Exp2 with Eval {
  case Sub(e1: Exp2.this, e2: Exp2.this) => this(e1) - this(e2)
}
```

Bibliography

[1] Sven Apel and Christian Lengauer. “Superimposition: A Language-Independent Approach to Software Composition”. In: *Software Composition*. Vol. 4954. Lecture Notes in Computer Science. Springer, 2008, pp. 20–35.

[2] Patrick Bahr. “Composing and decomposing data types: a closed type families implementation of data types à la carte”. In: *WGP@ICFP*. ACM, 2014, pp. 71–82.

[3] Patrick Bahr and Tom Hvitved. “Compositional data types”. In: *WGP@ICFP*. ACM, 2011, pp. 83–94.

[4] Aggelos Biboudis et al. “Streams a la carte: Extensible Pipelines with Object Algebras”. In: *ECOOP*. Vol. 37. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 591–613.

[5] Martin Bravenboer and Eelco Visser. “Parse Table Composition”. In: *SLE*. Vol. 5452. Lecture Notes in Computer Science. Springer, 2008, pp. 74–94.

[6] Martin Bravenboer et al. “Stratego/XT 0.17. A language and toolset for program transformation”. In: *Sci. Comput. Program.* 72.1-2 (2008), pp. 52–70.

[7] Peter Buchlovsky and Hayo Thielecke. “A Type-theoretic Reconstruction of the Visitor Pattern”. In: *Electr. Notes Theor. Comput. Sci.* 155 (2006), pp. 309–329.

[8] W.H. Burge. *Recursive Programming Techniques*. Addison-Wesley Series in Electrical and Computer Engineering. Addison-Wesley Longman, Incorporated, 1975. ISBN: 9780201144505.

[9] Eugene Burmako. “Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming”. In: *SCALA@ECOOP*. ACM, 2013, 3:1–3:10.

[10] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. “Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages”. In: *J. Funct. Program.* 19.5 (2009), pp. 509–543.

[11] Sergey Dmitriev. “Language oriented programming: The next programming paradigm”. In: *JetBrains onBoard* 1.2 (2004), pp. 1–13.

[12] *Dotty Issue 2029: Think about type class/implicits encoding*. URL: `https://github.com/lampepfl/dotty/issues/2029`.

[13] Burak Emir, Martin Odersky, and John Williams. “Matching Objects with Patterns”. In: *ECOOP*. Vol. 4609. Lecture Notes in Computer Science. Springer, 2007, pp. 273–298.

[14] *Ensō*. 2010. URL: `http://www.enso-lang.org`.

[15] Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. “Language composition untangled”. In: *LDTA*. ACM, 2012, p. 7.

[16] Erik Ernst. “Family Polymorphism”. In: *ECOOP*. Vol. 2072. Lecture Notes in Computer Science. Springer, 2001, pp. 303–326.

[17] Moritz Eysholdt and Heiko Behrens. “Xtext: implement your language faster than the quick and dirty way”. In: *SPLASH/OOPSLA Companion*. ACM, 2010, pp. 307–309.

[18] Jeroen Fokker. “Functional Parsers”. In: *Advanced Functional Programming*. Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 1–23.

[19] Bryan Ford. “Packrat parsing: simple, powerful, lazy, linear time, functional pearl”. In: *ICFP*. ACM, 2002, pp. 36–47.

[20] Martin Fowler. *Language workbenches: The killer-app for domain specific languages?* 2005. URL: https://martinfowler.com/articles/languageWorkbench.html.

[21] Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. “Parser Combinators for Ambiguous Left-Recursive Grammars”. In: *PADL*. Vol. 4902. Lecture Notes in Computer Science. Springer, 2008, pp. 167–181.

[22] *Full code of examples in Chapter 3*. URL: https://github.com/lihuanglx/tapl-visitor/tree/master/tapl/src/main/scala/language.

[23] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

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

[25] Robert Grimm. “Better extensibility through modular syntax”. In: *PLDI*. ACM, 2006, pp. 38–51.

[26] *Haskell mtl Package*. URL: https://hackage.haskell.org/package/mtl.

[27] John Hughes. “Why Functional Programming Matters”. In: *Comput. J.* 32.2 (1989), pp. 98–107.

[28] Graham Hutton. “Higher-Order Functions for Parsing”. In: *J. Funct. Program.* 2.3 (1992), pp. 323–343.

[29] Graham Hutton and Erik Meijer. “Monadic Parsing in Haskell”. In: *J. Funct. Program.* 8.4 (1998), pp. 437–444.

[30] Pablo Inostroza and Tijs van der Storm. “Modular interpreters for the masses: implicit context propagation using object algebras”. In: *GPCE*. ACM, 2015, pp. 171–180.

[31] Stephen C Johnson. *Yacc: Yet another compiler-compiler*. Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[32] Mark P. Jones. “Functional Programming with Overloading and Higher-Order Polymorphism”. In: *Advanced Functional Programming*. Vol. 925. Lecture Notes in Computer Science. Springer, 1995, pp. 97–136.

[33] Ulrik Jørring and William L. Scherlis. “Compilers and Staging Transformations”. In: *POPL*. ACM Press, 1986, pp. 86–96.

[34] Ted Kaminski et al. “Reliable and Automatic Composition of Language Extensions to C: The ableC Extensible Language Framework”. In: *OOPSLA*. ACM, 2017.

[35] Christian Kästner, Sven Apel, and Klaus Ostermann. “The road to feature modularity?” In: *SPLC Workshops*. ACM, 2011, p. 5.

[36] Lennart C. L. Kats and Eelco Visser. “The spoofax language workbench: rules for declarative specification of languages and IDEs”. In: *OOPSLA*. ACM, 2010, pp. 444–463.

[37] Steven Kelly, Kalle Lyytinen, and Matti Rossi. “MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment”. In: *CAiSE*. Vol. 1080. Lecture Notes in Computer Science. Springer, 1996, pp. 1–21.

[38] *kind-projector*. URL: `https://github.com/non/kind-projector`.

[39] Daan Leijen and Erik Meijer. *Parsec: Direct style monadic parser combinators for the real world*. Tech. rep. UU-CS-2001-27. Department of Computer Science, Universiteit Utrecht, 2001.

[40] Sheng Liang, Paul Hudak, and Mark P. Jones. “Monad Transformers and Modular Interpreters”. In: *POPL*. ACM Press, 1995, pp. 333–343.

[41] M. D. McIlroy. “Mass-produced software components”. In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968).

[42] Eugenio Moggi. *An abstract view of programming languages*. Tech. rep. ECS-LFCS-90-113. Laboratory for Foundations of Computer Science, University of Edinburgh, 1990.

[43] *MPS*. 2003. URL: `http://www.jetbrains.com/mps/`.

[44] Martin Odersky and Matthias Zenger. “Independently Extensible Solutions to the Expression Problem”. In: *Proc. FOOL 12*. 2005.

[45] Terence John Parr and Russell W. Quong. “ANTLR: A Predicated- *LL(k)* Parser Generator”. In: *Softw., Pract. Exper.* 25.7 (1995), pp. 789–810.

[46] Benjamin C. Pierce. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN: 0262162091, 9780262162098.

[47] Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. “From object algebras to attribute grammars”. In: *OOPSLA*. ACM, 2014, pp. 377–395.

[48] Tiark Rompf and Martin Odersky. “Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs”. In: *Commun. ACM* 55.6 (2012), pp. 121–130.

[49] Bruno C. d. S. Oliveira. “Modular Visitor Components”. In: *ECOOP*. Vol. 5653. Lecture Notes in Computer Science. Springer, 2009, pp. 269–293.

[50] Bruno C. d. S. Oliveira and William R. Cook. “Extensibility for the Masses - Practical Extensibility with Object Algebras”. In: *ECOOP*. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 2–27.

[51] Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. “Modular reifiable matching: a list-of-functors approach to two-level types”. In: *Haskell*. ACM, 2015, pp. 82–93.

[52] Bruno C. d. S. Oliveira et al. “Feature-Oriented Programming with Object Algebras”. In: *ECOOP*. Vol. 7920. Lecture Notes in Computer Science. Springer, 2013, pp. 27–51.

[53] *Scala Issue SI-1938: traits should be able to call super on fields*. URL: `https://issues.scala-lang.org/browse/SI-1938`.

[54] *scala-parser-combinators*. URL: `https://github.com/scala/scala-parser-combinators`.

[55] *Scalameta*. URL: `https://github.com/scalameta/scalameta`.

[56] *ScalaMeter*. URL: `https://github.com/scalameter/scalameter`.

[57] *Scalaz Issue 1110: MTL-style doesn't seem to work in Scala*. URL: `https://github.com/scalaz/scalaz/issues/1110`.

[58] August Schwerdfeger and Eric Van Wyk. “Verifiable composition of deterministic grammars”. In: *PLDI*. ACM, 2009, pp. 199–210.

[59] August Schwerdfeger and Eric Van Wyk. “Verifiable Parse Table Composition for Deterministic Parsing”. In: *SLE*. Vol. 5969. Lecture Notes in Computer Science. Springer, 2009, pp. 184–203.

[60] Wouter Swierstra. “Data types à la carte”. In: *J. Funct. Program.* 18.4 (2008), pp. 423–436.

[61] Mads Torgersen. “The Expression Problem Revisited”. In: *ECOOP*. Vol. 3086. Lecture Notes in Computer Science. Springer, 2004, pp. 123–143.

[62] Marcos Viera, S. Doaitse Swierstra, and Atze Dijkstra. “Grammar fragments fly first-class”. In: *LDTA*. ACM, 2012, p. 5.

[63] Tobi Vollebregt, Lennart C. L. Kats, and Eelco Visser. “Declarative specification of template-based textual editors”. In: *LDTA*. ACM, 2012, p. 8.

[64] Philip Wadler. “How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages”. In: *FPCA*. Vol. 201. Lecture Notes in Computer Science. Springer, 1985, pp. 113–128.

[65] Philip Wadler. “The Essence of Functional Programming”. In: *POPL*. ACM Press, 1992, pp. 1–14.

[66] Philip Wadler. “The expression problem”. In: *Java-genericity mailing list* (1998).

[67] Martin P Ward. “Language-oriented programming”. In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161.

[68] Alessandro Warth, James R. Douglass, and Todd D. Millstein. “Packrat parsers can support left recursion”. In: *PEPM*. ACM, 2008, pp. 103–110.

[69] Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. “Modular semantic actions”. In: *DLS*. ACM, 2016, pp. 108–119.

[70] Haoyuan Zhang, Huang Li, and Bruno C. d. S. Oliveira. “Type-Safe Modular Parsing”. In: *SLE*. ACM, 2017, forthcoming.

[71] Haoyuan Zhang et al. “Scrap your boilerplate with object algebras”. In: *OOP-SLA*. ACM, 2015, pp. 127–146.

[72] Weixin Zhang and Bruno C. d. S. Oliveira. “EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse”. In: *ECOOP*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 29:1–29:32.