Abstract of thesis entitled

“Modularity Beyond Compositionality”

Submitted by

Weixin Zhang

for the Degree of Doctor of Philosophy
at The University of Hong Kong
in April 2021

Modern software becomes increasingly complex and rapid-evolving. The cost of developing and maintaining software systems is extremely high. To reduce the engineering effort involved in developing and maintaining software systems, modularity plays an important role. A modular way to develop software will subdivide a software system into a bunch of smaller components. Each of those components can then be of a manageable size, can be developed in parallel with isolated functionality assigned to specific programmers. The components should be extensible so that new functionality can be added easily without affecting other existing components. However, such a modular extensible style of component-based software development is not widely applicable in practice due to inadequate support from the modularization techniques in programming languages.

This thesis aims at investigating techniques on programming languages that support modular extensible software development from three perspectives: plain design patterns, metaprogramming-based design patterns, and novel language designs. Plain design patterns can be directly applied in existing programming languages. Existing languages have an advantage that programmers are already familiar with, allowing a wider application of the modularization technique without a steep learning curve. With the help of metaprogramming, the boilerplate code associated with the design patterns can be automated. However, existing languages are sometimes too restrictive in terms of syntax and semantics, which might not be expressive or concise enough for certain cases. For such cases, new programming languages (features) suit better.

The first part of the thesis focuses on plain design patterns. We connect shallow embeddings to Object-Oriented Programming (OOP) via procedural abstraction. We argue that common OOP mechanisms increase the modularity and reuse of shallow EDSLs. We make our arguments by using Gibbons and Wu’s examples, where procedural abstraction is used in Haskell to model a simple shallow EDSL. We recode that EDSL in Scala and with an improved OO-inspired Haskell encoding. We further illustrate our approach with a case study on refactoring a deep external SQL query processor to make it more modular, shallow, and embedded.

The second part of the thesis turns to metaprogramming-based design patterns. We present Castor, a Scala framework for programming with extensible, generative visitors. Castor has several advantages over previous approaches including its support for pattern matching, type-safe interpreters, imperative style visitors, and graphs. The applicability of Castor is shown by several examples and two case studies on modularizing interpreters from the "Types and Programming Languages" book and UML activity diagrams.

The last part of the thesis is on novel language designs. We propose a programming style called Compositional Programming. We introduce four key concepts for Compositional Programming: compositional interfaces, compositional traits, method patterns, and nested trait composition. Altogether these concepts allow us to naturally solve challenges such as the Expression Problem, model attribute-grammar-like programs, and generally deal with modular programs with complex dependencies. We present a language design, called CP, which is proved to be type-safe, together with several examples and three case studies.

(483 words)

THE UNIVERSITY OF HONG KONG

Modularity Beyond Compositionality

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

Weixin Zhang

April 2021

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.

. . . .

Weixin Zhang

April 2021

Acknowledgements

First of all, I owe sincere thanks to my brilliant advisor Dr. Bruno C. d. S. Oliveira for a long time. This thesis would never be done without his continuous help and guidance. Bruno is such a great role model of a researcher whose enthusiasm for research will have a lifelong influence on me.

It was a long and special journey at HKU. Along the way, I received a lot of help from many people. The beginning was not easy for me both financially and mentally. Thank Prof. T.H. Tse for generously supporting me as a research assistant during part of my MPhil period. Thank Bruno for allowing me to switch to another research project when I got stuck. Thank staffs from the general office of Department of Computer Science for their help, in particular: Priscilla Chung, Olive Hui, and Maria Lam.

I would like to thank the members of HKU PL Group for the insightful discussions and constructive comments on my work: Xuan Bi, Haoyuan Zhang, Yanlin Wang, Yanpeng Yang, Tomas Tauber, Zhiyuan Shi, Ningning Xie, Huang Li, Jinxu Zhao, Xuejing Huang, Yaoda Zhou, Baber Rehman, Yaozhu Sun, to name a few.

I would also like to thank other friends in Hong Kong: Qiuling Zeng, Yanmin Zhao, Xuhui Jia, Rui Zhou, Kai Han, Yu Liu, Ningyu Liu, and Kan Wu.

I am grateful to live in an era where MOOCs are free to everyone. Thank Prof. Dan Grossman, Prof. Martin Odersky, and Prof. Gregor Kiczales for their fantastic MOOCs that stimulated my interest in programming languages.

When I doubted myself and felt uncertain about the direction after a PhD, Dr. Meng Wang encouraged me and offered me a postdoc position. Thank Meng for being an excellent host when I was a stranger in Bristol. I am fortunate to have Dr. Tijs van der Storm and Dr. Marco Servetto been my referees. I appreciate the fruitful discussions with them at several conferences.

Thank my parents and family members for their endless love and support for me.

Finally, I hope the COVID-19 pandemic will end and the world will back to normal soon!

April 2021

Weixin Zhang

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why Modularity Matters

Ever since the software crisis [Naur, 1968], modern software systems have become increasingly complex, large-scale and rapid-evolving. The cost of developing and maintaining software systems is extremely high. To reduce the engineering effort involved in developing and maintaining software systems, plays an important role in software development. A modular way to develop software will subdivide a software system into a bunch of smaller components. Each of those components can then be of a manageable size, can be developed in parallel with isolated functionality assigned to specific programmers. The functionality of each component is exposed by a well-defined interface with which other components communicate. Interacting through interfaces ensures loose coupling between the components and allows multiple implementations of the same interface to coexist and be readily swapped. The components should be extensible so that new functionality can be added easily without affecting other existing components. Developed in a modular way, these components can further serve as reusable units for developing software systems that have similar functionality, forming a software product line [Pohl et al., 2005]. Yet, such a modular style of component-based software development [Heineman and Councill, 2001] is not widely applicable in practice due to the inadequate support from the modularization techniques in programming languages.

To enable the software development style described above, a proper modularization technique should meet the following requirements:

- *Extensibility in both dimensions:* New data variants and new operations can be added to the system;

- *Strong static type safety:* The mismatch between variants and operations can be statically caught by the type system;

- *No modification or duplication:* Existing components can neither be modified nor duplicated;

- *Separate compilation and type-checking:* Existing components should neither be type-checked nor compiled again when adding extensions;

- *Independent extensibility:* Independently developed extensions can be combined for joint use;
- *Modular dependencies:* Components can depend on other components and such dependencies can be expressed modularly.

The first four requirements come from the well-known modularity challenge – the *Expression Problem* (EP) [Wadler, 1998]. The EP examines the *type-safe extensibility* of programming languages, where extensions should be done without rewriting existing code and without using type-unsafe features (such as casts or reflection). Solving the EP is important for several reasons. Firstly, the two dimensions of extensibility (data variants and operations) capture one essential aspect of software evolution. Secondly, strong type-safety avoids runtime errors and ensures the safety of component compositions. Thirdly, without duplication, the code is easier to maintain. Modification is not even possible when the source code may not be available such as distributed in binary forms. Lastly, the cost of compilation and type-checking is high for complex software systems. Separate compilation and type-checking boost the development process. The fifth requirement is added by Zenger and Odersky [2005], which addresses the *compositionality*. Compositionality is important for allowing separately developed components to be merged in the final system. We further propose the sixth requirement that assesses the *dependability*. Dependability is critical because realistic programs will depend on some other program parts. A common case of program dependencies is using library functions or functions defined in other parts of the program. It is important to have modular dependencies because the weaker dependencies between program parts are, the more modular a program is.

1.2 Problem Statement

Small canonical modularity problems, such as the EP, illustrate some of the basic issues: the dilemma between choosing one kind of extensibility over another one in most programming languages. However, there are more modularity challenges not covered by the EP, but also very important in practice. In particular, an important issue and a focus of this thesis are how to express dependencies modularly. Therefore, extending the EP, we set up a concrete problem for examining the ability of programming languages on modularity.

1.2.1 The Expression Problem

The initial setting of the EP is about how to create a very simple form of expressions (such as numeric literals, addition, and multiplication), and operations over those expressions (such as evaluation and pretty-printing) in a modular way.

Conventional Object-Oriented Programming (OOP) and functional programming (FP) cannot solve the EP since they typically excel at only one dimension of extensibility. We now illustrate the dilemma in OOP and FP using the Scala language [Odersky et al., 2004] since it supports both object-oriented and functional paradigms.

```
trait Exp {
  def eval:  Int
}
class Lit(n: Int) extends Exp {
  def eval  = n
}
class Add(e1: Exp, e2: Exp) extends Exp {
  def eval  = e1.eval + e2.eval
}
// Adding new variants is easy!
class Mul(e1: Exp, e2: Exp) extends Exp {
  def eval = e1.eval * e2.eval
}
```

Figure 1.1: *Object-oriented programming style*

```
sealed abstract class Exp
case class Lit(n: Int) extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp

def eval(e: Exp): Int = e match {
  case Lit(n)      => n
  case Add(e1, e2) => eval(e1) + eval(e2)
}
// Adding new operations is easy!
def print(e: Exp): String = e match {
  case Lit(n)      => n.toString
  case Add(e1, e2) => "(" + print(e1) + "+" + print(e2) + ")"
}
```

Figure 1.2: *Functional programming style*

Object-oriented programming Figure 1.1 gives an OOP implementation of the simple expression language. Expressions are modeled as a class hierarchy, and operations on the datatype are methods implemented throughout the hierarchy. It is quite easy to add more data variants (such as multiplication) by creating new subclasses of `Exp`. However, adding new operations (such as pretty-printing) is difficult because every class needs amendments for new methods. Such amendments violate the *open-closed principle* in OOP [Meyer, 1988].

Functional programming The situation is exactly the opposite when it comes to FP. Figure 1.2 gives an FP-style implementation of the simple expression language. Expressions are modeled using an algebraic data type (a **case** class hierarchy in Scala) and operations over expressions are defined as pattern matching functions. It is easy to add new operations just by defining new pattern matching functions. However, adding data variants becomes difficult since it requires modification to every function for new patterns.

1.2.2 Dependent Operations

An important modularity challenge not covered by the EP is about dependencies [Oliveira et al., 2013; Rendel et al., 2014; Zhang and Oliveira, 2017]. The following code snippet shows a simple form of dependency, where one operation depends on another operation:

```
def dprint(e: Exp): String = e match {
  case Lit(n)      => n.toString
  case Add(e1, e2) =>
    if (eval(e1) == 0) dprint(e2) // dependency on eval
    else "(" + dprint(e1) + "+" + dprint(e2) + ")"
}
```

dprint depends on eval: if the first operand of an addition expression evaluates to zero, only its second operand will be printed. dprint expresses such dependency by directly referring to eval. Such a way of declaring dependency is not modular because it ties dprint to a fixed implementation of evaluation. How to express the dependency modularly imposes new challenges to modularization techniques.

1.2.3 Binary and Producer Operations

Binary and producer operations [Bruce et al., 1995] are also challenges not covered by the EP. Binary operations take two parameters of a datatype while producer operations return values of a datatype. Binary and producer operations are problematic in a modularity setting because the datatype they refer to might be extended.

Binary operations A typical example of binary operations is structural equality. Structural equality further examines the ability to destruct data structures. An FP-style implementation of structural equality is given below:

```
def equals(e1: Exp, e2: Exp): Boolean = (e1,e2) match {
  case (Lit(v1), Lit(v2))       => v1 == v2
  case (Add(e1,e2), Add(e3,e4)) => equals(e1,e3) && equals(e2,e4)
  case _                        => false
}
```

equals compares whether two expressions are structurally the same by pattern matching on the two expressions simultaneously. However, implementing equals in an OOP-style is tricker:

```
trait Exp {
  def equals(that: Exp): Boolean
}
class Lit(val n: Int) extends Exp {
  def equals(that: Exp) =
    if (that.isInstanceOf[Lit])
      n == that.asInstanceOf[Lit].n
    else
      false
}
class Add(val e1: Exp, val e2: Exp) extends Exp {
  def equals(that: Exp) =
    if (that.isInstanceOf[Add]) {
      val add = that.asInstanceOf[Add]
      e1.equals(add.e1) && e2.equals(add.e2)
    }
    else
      false
}
```

The implementation is verbose and type-unsafe since type-tests and type-casts are used for recognizing the concrete representation of `Exp`.

Producer operations Besides referring to datatypes, producer operations may also call the constructors for rebuilding data structures, which is another challenge for modularity. An instance of producer operation is given below:

```
def evalToLit(e: Exp): Exp = e match {
  case Add(Lit(n1),Lit(n2)) => Lit(n1+n2)
  case Add(e1,e2)           => evalToLit(Add(evalToLit(e1),evalToLit(e2)))
  case _                    => e
}
```

`evalToLit` is another version of evaluation that recursively rewrites an expression into a literal, where constructors `Lit` and `Add` are called for rewriting the expression.

1.3 Approaches to Modular Extensibility

A lot of work has been done on modular extensibility in both software engineering and programming languages communities over the past several decades. We classify the approaches to modular extensibility into two categories: *syntactic* modularity approaches and *semantic* modularity approaches. Syntactic modularity approaches allow the code to be separately defined and then concatenated as a whole for type-checking and compilation. There are some drawbacks of syntactic approaches. Existing code has to be type-checked and compiled again together with the newly added code, which is time-consuming especially for complex software systems. Moreover, the error messages are reported not on the original code but on the generated code, which may be hard to understand. Compared to syntactic modularity approaches, semantic modularity approaches preserve separate compilation and modular typechecking so that the problems abovementioned can be avoided.

1.3.1 Syntactic Modularity Approaches

Software development/Programming paradigms Many software development/programming paradigms have been proposed for tackling different aspects of modularity problems. Well-known approaches like *subject-oriented programming* [Harrison and Ossher, 1993], *component-based software development* [Heineman and Councill, 2001], *multi-dimensional separation of concerns* [Tarr et al., 1999], *aspect-oriented programming* [Kiczales et al., 1997] and *feature-oriented programming* [Prehofer, 1997] offer different perspectives on how to decompose programs modularly and deal with challenges such as cross-cutting concerns. These approaches can be classified as syntactic modularity approaches because the composition mechnism provided by these approaches is typically realized using metaprogramming.

1.3.2 Semantic Modularity Approaches

Programming languages Programming languages provide language constructs for modularization. For instance, many languages support some notion of *modules* [MacQueen, 1984] that can group various kinds of definitions and functions and can be separately compiled. At a smaller scale, most OOP languages support *classes* which can also be separately defined, compiled, and reused by subclassing. Suffering from the diamond problem, traditional classes support only single inheritance, restricting the reusability. *Mixins* [Bracha and Cook, 1990] handle the diamond problem through linearization where conflicts between mixins are implicitly resolved according to the composition order, allowing a form of multiple inheritance. However, a disadvantage of such an order-based resolution mechanism is that selecting conflicting implementations from multiple sources becomes impossible. Without dealing with state, *traits* [Schärli et al., 2003] serve as lightweight reusable units, where conflicts must be explicitly resolved by the programmer.

Ideas such as *virtual classes* [Madsen and Moller-Pedersen, 1989; Ernst et al., 2006] and *family polymorphism* [Ernst, 2001], extend the idea of virtual methods to classes. Thus, *classes* and *constructors* can themselves be *virtual*, weakening the dependencies to classes and constructors. Family polymorphism provides powerful forms of reuse, which can solve tricky modularity problems, such as the EP, naturally [Ernst, 2004; Ernst et al., 2006]. Nevertheless, family polymorphism still uses inheritance as a primary mechanism to express dependencies. Similarly to (regular) classes, the use of inheritance in family polymorphism sometimes creates more coupling than necessary between sub- and superclasses/families. Furthermore, the type systems and semantics for calculi with virtual classes are quite complex and often require advanced features such as forms of *dependent types* [Ernst et al., 2006; Nystrom et al., 2006, 2004].

Other work has focused on more general language features – such as *generics*, *higher-kinded types* [Moors et al., 2008], *virtual types* [Thorup, 1997] – which can also help with various modularity problems.

Design patterns Although language features specialized for modularity are very powerful, they are typically not available in mainstream programming languages. Much of the more recent work on type-safe extensibility focus on design patterns [Gamma et al., 1994] that are applicable to mainstream languages with modest type system features. Traditional design patterns, in particular the Interpreter pattern and the Visitor pattern, support only one dimension of extensibility. Improving on traditional design patterns, new design patterns such as *Extensible Visitors* [Torgersen, 2004; Oliveira, 2009; Hofer and Ostermann, 2010; Zhang and Oliveira, 2017], *Extensible Interpreters* [Torgersen, 2004; Wang and Oliveira, 2016], *Object Algebras* [Oliveira and Cook, 2012], *Finally Tagless in-*

terpreters [Carette et al., 2009] or *Polymorphic Embeddings* [Hofer et al., 2008] are capable of solving the EP.

Essentially these techniques are closely related. The foundation for a lot of that work comes from functional programming and type-theoretic encodings of datatypes [Church, 1936; Scott, 1963]. In particular, the work by Hinze [2006] was the precursor for those techniques. In his work Hinze employed so-called Church [1936] and Scott [1963] encodings of datatypes to model generic programming libraries. Later Oliveira et al. [2006a] showed that variants of those techniques have wider applications and solve the EP. These ideas were picked up by Carette et al. [2009] to enable tagless interpreters, while also benefiting from the extensibility properties of the techniques. Carette et al.'s [2009] work popularized those applications of the techniques as the nowadays so-called *Finally Tagless* style. Soon after Hofer et al. [2008] proposed *Polymorphic Embeddings* in Scala, highly inspired by the Finally Tagless style in languages like Haskell and OCaml.

In parallel with the work on *Finally Tagless* and *Polymorphic Embeddings* the connections of those techniques to the Visitor pattern in OOP were further explored, building on observations between the relationship between type-theoretic encodings of datatypes and visitors by Buchlovsky and Thielecke [2006]. That work showed that Church and Scott encodings of datatypes correspond to two variants of the Visitor pattern called, respectively, *Internal* and *External* visitors. Later on Oliveira and Cook [2012] showed a simplified version of *Internal Visitors* called *Object Algebras*, which could solve the EP even in languages like Java.

While *Internal Visitors*, *Object Algebras*, *Finally Tagless* or *Polymorphic Embeddings* can all be traced back to Church encodings, there has been much less work on techniques that are based on Scott encodings. Scott encodings are more powerful, as they allow a (generally) recursive programming style. In contrast, Church encodings rely on a programming style that is akin to programming with folds in functional programming [Gibbons, 2003]. In general, Scott encodings require more sophisticated type system features, which is one reason why they have seen less adoption. In particular recursive types are necessary, which also brings up extra complications due to the interaction of recursive types and subtyping.

However, due to the lack of proper language support, these design patterns often result in heavily parametrized and boilerplate code. Moreover, the lack of sufficiently powerful composition mechanisms makes dealing with dependencies hard in such design patterns [Zhang and Oliveira, 2017; Oliveira et al., 2013].

1.4 Contributions

This thesis aims at investigating techniques for modular extensibility in existing programming languages as well as new programming languages (features) that, to a large extent, meet the requirements summarized in Section 1.1. Existing languages have an advantage that programmers are already familiar with, allowing a wider application of the modularization technique without a steep learning curve. However, existing languages are sometimes too restrictive in terms of syntax and semantics, which might not be expressive or concise enough for certain cases. For such cases, new programming languages (features) suit better. Therefore, it is meaningful to investigate modularization techniques in both existing and new programming languages. In summary, solutions proposed in this thesis are classified in three different styles:

1. *Plain design patterns*, which can be directly applied in existing programming languages;

2. *Metaprogramming-based design patterns*, where the bolierplate code incurred by design patterns is further automated;

3. *Novel language designs*, which allows specialized syntax and semantics designed for advanced modularity.

Shallow EDSLs and OOP The first part of the thesis focuses on plain design patterns. The modularity issue also occurs in embedded domain-specific languages (EDSLs). The two approaches to EDSLs, shallow embeddng and deep embedding, complement with each other in terms of extensibility: adding new variants is easy in shallow embeddings while adding new interpretations is easy in deep embeddings. This makes it hard for programmers to choose between the two embeddings when the extension dimension is unknown in advance. Gibbons and Wu [2014] already discussed the relationship between shallow EDSLs and procedural abstraction, while Cook [2009] discussed the connection between procedural abstraction and OOP. We make the transitive step by connecting shallow EDSLs directly to OOP via procedural abstraction. The knowledge about this relationship enables us to improve on implementation techniques for EDSLs. Common OOP mechanisms (including *subtyping*, *inheritance*, and *type-refinement*) increase the modularity and reuse of shallow EDSLs when compared to classical procedural abstraction by enabling a simple way to express *multiple, possibly dependent, interpretations*. We make our arguments by using Gibbons and Wu's examples, where procedural abstraction is used in Haskell to model a simple shallow EDSL. We recode that EDSL in Scala using the *Extensible Interpreter* pattern [Wang and Oliveira, 2016] and with an improved OO-inspired Haskell encoding. We further illustrate our approach with a case study on refactoring a deep external SQL query processor [Rompf and Amin, 2015] to make it more modular, shallow, and embedded.

The Castor framework The second part of the thesis turns to metaprogramming-based design patterns. We develop Castor, a Scala framework for programming with extensible, generative visitors. Castor has several advantages over previous approaches. Firstly, Castor comes with support for (type-safe) pattern matching to complement its visitors with a concise notation to express operations. Secondly, Castor supports type-safe interpreters (à la Finally Tagless), but with additional support for pattern matching and a generally recursive style. Thirdly, Castor enables many operations to be defined using an imperative style, which is significantly more performant than a functional style (especially in the JVM platform). Finally, functional techniques usually only support tree structures well, but graph structures are poorly supported. Castor supports type-safe extensible programming on graph structures. The key to Castor's usability is the use of annotations to automatically generate large amounts of boilerplate code to simplify programming with extensible visitors. To illustrate the applicability of Castor we present several applications and two case studies. The first case study compares the ability of Castor for modularizing the interpreters from the "Types and Programming Languages" book [Pierce, 2002] with previous modularization work. The second case study on UML activity diagrams illustrates the imperative aspects of Castor, as well as its support for hierarchical datatypes and graphs.

Compositional Programming In the last part of the thesis, we concentrate on novel language designs. We propose a programming style called *Compositional Programming* that encourages: 1) references to methods (or functions), constructors and types to be virtual or abstract; and 2) dependencies to other program components to be expressed in terms of interfaces, instead of concrete implementations. There are four key concepts in Compositional Programming. Compositional interfaces extend conventional OOP interfaces to allow the specification of the signatures of constructors, and can be parametrized by sorts (which abstract over concrete data types). Compositional traits extend (first-class) traits to allow not only the definition of (virtual) methods but also the definition of virtual constructors. Method patterns provide a lightweight syntax to define implementations for nested traits, which arise from virtual constructors. Finally, a powerful form of nested trait composition is used to compose compositional traits. Nested trait composition plays a similar role to traditional class inheritance, but it generalizes to the composition of nested traits. Thus it enables a form of inheritance of whole hierarchies, similar to the forms of composition found in family polymorphism. Altogether these concepts allow us to naturally solve challenges such as the EP, model attribute-grammar-like programs, and generally deal with modular programs with complex dependencies. We present several examples and three case studies. Our first case study is on the design of an EDSL for circuits [Hinze, 2004; Gibbons and Wu, 2014]. This EDSL is interesting because it has various extensions that can be modularly defined, as well as various dependencies between components. Our second case study is a mini interpreter, which is larger and can be extended in several ways. The last case study is an implementation of the C0 compiler, inspired by the work of Rendel et al. [2014]. In this case study, various extensions can be formulated as attributes, and those attributes contain non-trivial dependencies to other attributes. Finally, we present a small calculus that captures the essence of CP. This calculus is shown to be type-safe via an elaboration to the F_i^+ calculus [Bi et al., 2019], which is a recently proposed calculus that supports *disjoint intersection types* [Oliveira et al., 2016], *disjoint polymorphism* [Alpuim et al., 2017] and *nested composition* [Bi et al., 2018].

1.5 Organization

The rest of this thesis is organized as follows:

- Chapter 2 gives the necessary background of the thesis;
- Chapter 3 connects shallow embeddings to OOP and show how OOP features improve on the modularity of shallow EDSLs;
- Chapter 4 presents the Castor framework and various applications of Castor;
- Chapter 5 presents the Compositional Programming style and the CP language designed for it;
- Chapter 6 discusses the related work;
- Chapter 7 discusses the future work;
- Chapter 8 concludes the thesis.

This thesis is based on the previous publications by the author [Zhang and Oliveira, 2018, 2019, 2020; Zhang et al., 2021] and the initial idea of Chapter 3 comes from the author's MPhil thesis [Zhang, 2017].

Chapter 2

Preliminaries

This chapter gives the preliminaries of the thesis. The techniques reviewed in this chapter, including the extensible Interpreter pattern [Wang and Oliveira, 2016], the Visitor pattern [Gamma et al., 1994], the Cake pattern [Odersky and Zenger, 2005], Object Algebras [Oliveira and Cook, 2012; Oliveira et al., 2013], disjoint intersection types [Oliveira et al., 2016; Alpuim et al., 2017; Bi et al., 2018, 2019] and SEDEL [Bi and Oliveira, 2018], are either directly used in or improved by this thesis.

2.1 The Extensible Interpreter Pattern

The Interpreter pattern [Gamma et al., 1994], as its name suggests, is often used in implementing interpreters for programming languages. The OOP code shown in Figure 1.1 essentially follows the conventional Interpreter pattern, which is easy to add new variants but hard to add new operations. In fact, as shown by Wang and Oliveira [2016], it is possible to make the Interpreter extensible with common OOP features such as *subtyping*, *inheritance* and *type-refinement*. The code shown in Figure 1.1 can be revised using the extensible interpreter pattern:

```
trait Exp {
  def eval: Int
}
trait Lit extends Exp {
  val n: Int
  def eval = n
}
trait Add extends Exp {
  val e1, e2: Exp
  def eval = e1.eval + e2.eval
}
```

where traits are used instead of classes in implementing `Exp`.

Then dependent printing method can be modularly added:

```
trait ExpExt extends Exp { // subtyping
  def print: String
}
trait LitExt extends Lit with ExpExt { // inheritance
  def print = n.toString
}
```

```
trait AddExt extends Add with ExpExt { // inheritance
  val e1, e2: ExpExt // type-refinement
  def print =
    if (e1.eval == 0) e2.print
    else "(" + e1.print + "+" + e2.print + ")"
}
```

Another hierarchy is defined, where the root `ExpExt` is a *subtype* of `Exp`, `Lit` and `Add` are *inherited* in implementing the extended hierarchy and fields of type `Exp` are refined to `ExpExt` for allowing invocations on both `eval` and `print`.

We can construct expressions using the extended hierarchy:

```
val e = new AddExt {
  val e1 = new LitExt { val n = 0 }
  val e2 = new LitExt { val n = 1 }
}
println(e.print) // "1"
```

2.2 The Visitor Pattern

The Visitor pattern [Gamma et al., 1994] is often used interchangeably with the Interpreter pattern in implementing interpreters/compilers. The Visitor pattern is closely related to the functional approach shown in Figure 1.2, which is easy to add new operations but hard to add new variants. Let us implement the expression language step by step using the Visitor pattern.

Class hierarchy The expressions are modeled as an ordinary class hierarchy with an `accept` method implemented throughout the hierarchy.

```
trait Exp {
  def accept[E](v: ExpVisit[E]): E
}
class Lit(val n: Int) extends Exp {
  def accept[E](v: ExpVisit[E]) = v.lit(this)
}
class Add(val e1: Exp, val e2: Exp) extends Exp {
  def accept[E](v: ExpVisit[E]): E = v.add(this)
}
```

The `accept` method is implemented by calling the corresponding visit method exposed by an `ExpVisit` object.

Visitor interfaces `ExpVisit` is the visitor interface whose definition is as follows:

```
trait ExpVisit[E] {
  def lit(x: Lit): E
  def add(x: Add): E
}
```

Each subclass of `Exp` has a corresponding visit method, which takes an instance of that class and returns `E`.

Concrete visitors Then the evaluation operation is defined as a concrete visitor that implements the visitor interface ExpVisit:

```
object eval extends ExpVisit[Int] {
  def lit(x: Lit) = x.n
  def add(x: Add) = x.e1.accept(this) + x.e2.accept(this)
}
```

The concrete visitor eval implements ExpVisit by instantiating the type parameter E as Int and defining lit and add accordingly. To apply the visitor recursively to the inner expressions, **this** is passed as an argument to accept.

Adding new operations is done by defining new concrete visitors:

```
object dprint extends ExpVisit[String] {
  def lit(x: Lit) = x.n.toString
  def add(x: Add) =
    if (x.e1.accept(eval) == 0) x.e2.accept(this)
    else "("+ x.e1.accept(this) + "+" + x.e2.accept(this) + ")"
}
```

eval is directly used by passing eval as argument to accept calls.

We can construct an expression and print it like this:

```
val e = new Add(new Lit(0), new Lit(1))
println(e.accept(dprint)) // "1"
```

Unfortunately, without modifying the existing code, it is hard to add a new subclass such as Mul to the Exp hierarchy because there is no corresponding visit method exposed by ExpVisit for implementing its accept method.

2.3 The Cake Pattern

It is possible to make the conventional Visitor pattern extensible. The key is to decouple the class hierarchy from a specific visitor interface. As shown by Zenger and Odersky [2005], the fundamental ingredients of the Cake pattern [Odersky and Zenger, 2005], namely *abstract type members (virtual types)*, *self-type annotations* and *mixin composition*, provide a solution. We adapt their extensible imperative visitor encoding into a functional one and apply it in modeling the expression language:

```
trait Base {
  type ExpV[E] <: ExpVisit[E]
  trait Exp {
    def accept[E](v: ExpV[E]): E
  }
  trait ExpVisit[E] {
    def lit(x: Lit): E
    def add(x: Add): E
  }
  class Lit(val n: Int) extends Exp {
    def accept[E](v: ExpV[E]) = v.lit(this)
  }
  class Add(val e1: Exp, val e2: Exp) extends Exp {
    def accept[E](v: ExpV[E]) = v.add(this)
  }
  trait Eval extends ExpVisit[Int] { _: ExpV[Int] =>
    def lit(x: Lit) = x.n
    def add(x: Add) = x.e1.accept(this) + x.e2.accept(this)
  }
  trait DPrint extends ExpVisit[String] { _: ExpV[String] =>
    def lit(x: Lit) = x.n.toString
    def add(x: Add) =
      if (x.e1.accept(eval) == 0) x.e2.accept(this)
      else "(" + x.e1.accept(this) + "+" + x.e2.accept(this) + ")"
  }
  val eval: ExpV[Int]
}
```

There are several changes to the implementation using conventional visitors. Firstly, the definitions are put into a trait `Base`. Secondly, a higher-kinded, bounded abstract type member `ExpV` is introduced for decoupling the `Exp` hierarchy from the visitor interface `ExpVisit`. `ExpV` is constrained as a subtype of `ExpVisit` so that visit methods declared by `ExpVisit` can still be invoked when implementing the `accept` methods. Thirdly, visitors are defined as traits rather than objects. To pass **`this`** as an argument for recursive `accept` calls, the visitor has a self-type annotation as `ExpV`. Lastly, to express the dependency on evaluation, a field `eval` of type `Eval` is introduced.

Extensions can be done by extending `Base`:

```
trait Extension extends Base {
  type ExpV[E] <: ExpVisit[E]
  class Mul(val e1: Exp, val e2: Exp) extends Exp {
    def accept[E](v: ExpV[E]) = v.mul(this)
  }
  trait ExpVisit[E] extends super.ExpVisit[E] {
    def mul(x: Mul): E
  }
  trait Eval extends ExpVisit[Int] with super.Eval { _: ExpV[Int] =>
    def mul(x: Mul) = x.e1.accept(this) * x.e2.accept(this)
  }
  trait DPrint extends ExpVisit[String] with super.DPrint { _: ExpV[String] =>
    def mul(x: Mul) = x.e1.accept(this) + "*" + x.e2.accept(this)
  }
}
```

A new subclass of `Exp`, `Mul`, is modularly added to the hierarchy. To implement the `accept` method of `Mul`, `ExpVisit` is extended with a visit method `mul` and the `ExpV` is covariantly refined to the extended `ExpVisit`. Existing visitors, `Eval` and `DPrint`, are extended with an implementation of `mul`.

`Extension` needs to be instantiated before use:

```
object Extension extends Extension {
  type ExpV[E] = ExpVisit[E]
  object eval extends Eval
  object dprint extends DPrint
}
```

where `ExpV` binds to `ExpVisit` and each visitor has a lowercase instantiated object.

Finally we can construct an expression and apply operations to it:

```
import Extension._
val e = new Mul(new Lit(2), new Add(new Lit(0), new Lit(1)))
println(e.accept(dprint))
```

2.4 Object Algebras

Object Algebras [Oliveira and Cook, 2012] are a well-known OOP solution to the EP. The abstract syntax of the expression language is described by an Object Algebra interface (a Scala trait):

```
trait ExpAlg[Exp] {
  def Lit: Int       => Exp
  def Add: (Exp,Exp) => Exp
}
```

Essentially `ExpAlg` is an *abstract factory*, where the type parameter `E` captures the expression type and capitalized methods are factory methods returning variants of expressions.

Operations over the expressions are *concrete factories* that implement the `ExpAlg` interface. For example, evaluation can be defined as:

```
trait IEval {
  def eval: Int
}
trait Eval extends ExpAlg[IEval] {
  def Lit = n => new IEval {
    def eval = n
  }
  def Add = (e1,e2) => new IEval {
    def eval = e1.eval + e2.eval
  }
}
object eval extends Eval
```

`Eval` implements `ExpAlg` by instantiating `E` as `IEval` and returning an instance of `IEval` for each factory method accordingly.

Adding new operations It is easy to add new operations by reimplementing the `ExpAlg` interface. For example, printing is defined in a way similar to evaluation:

```
trait IPrint {
  def print: String
}
trait Print extends ExpAlg[IPrint] {
  def Lit = n => new IPrint {
    def print = n.toString
  }
  def Add = (e1,e2) => new IPrint {
    def print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")"
  }
}
object print extends Print
```

where `Print` instantiates `E` as `IPrint` and implements each factory method accordingly.

Adding new variants It is also easy to add new variants by extending the Object Algebra interface with new factory methods. For example, multiplications are modularly introduced as follows:

```
trait MulAlg[Exp] extends ExpAlg[Exp] {
  def Mul: (Exp,Exp) => Exp
}
```

where `MulAlg` extends `ExpAlg` with a new constructor `Mul`. Existing operations such as `Eval` can be modularly reuse in extensions:

```
trait EvalMul extends MulAlg[IEval] with Eval {
  def Mul = (e1,e2) => new IEval {
    def eval = e1.eval * e2.eval
  }
}
```

where `EvalMul` inherits `Eval` and complements the definition for `Mul` only.

Modular terms Now we show how to modularly construct a simple addition expression:

```
def exp[E](f: ExpAlg[E]) =
  f.Add(f.Lit(0),f.Lit(1))
```

The generic function `exp` builds an addition expression via the constructors exposed by the abstract algebra `f` Concrete expressions can be obtained by calling `exp` with concrete algebras such as `eval` and `print`:

```
println(exp(eval).eval)
println(exp(print).print)
```

By supplying `Eval` and `Print`, the expression can be respectively evaluated and printed.

2.4.1 Limitations of Object Algebras

Object Algebras, in their basic form, have several limitations in terms of *compositionality* and *dependability*. To address these issues, Oliveira et al. [2013] generalize Object Algebras and employ Scala's support for intersection types. We discuss the limitations one by one and show how generalized Object Algebras try to address these limitations.

Object Algebra combinators The first issue is that *there is no proper composition mechanism for Object Algebras*. In the previous section, the expression is indeed constructed *twice* respectively for evaluating and printing. A more effcient way is to compose `Eval` and `Print` into a single algebra so that the expression can be constructed *only once* for both evaluating and printing. A workaround is to use pair-based Object Algebra combinators [Oliveira and Cook, 2012], which require explicit projections and hence are inconvenient for use. Fortunately, explicit projections can be eliminated with the help of intersection types. In Scala, the type `A with B` denotes the intersection of two types `A` and `B`, where `A with B` is a subtype of both `A` and `B`. The intersection-based Object Algebra combinator for `ExpAlg` can be defined as:

```
trait ExpMerge[A,B] extends ExpAlg[A with B] {
  val alg1: ExpAlg[A]
  val alg2: ExpAlg[B]
  def lift(x: A, y: B) : A with B

  def Lit = n =>
    lift(alg1.Lit(n),alg2.Lit(n))
  def Add = (e1,e2) =>
    lift(alg1.Add(e1, e2),alg2.Add(e1, e2))
}
```

ExpMerge captures the two algebras to be composed as fields alg1 and alg2. ExpMerge implements ExpAlg by instantiating the type parameter as A **with** B and defining each factory method by firstly calling the corresponding factory method respectively defined on alg1 and alg2 then merging the results via the lift method. Concrete composition is done by implementing ExpMerge with alg1, alg2 and lift defined. For example, the composition of Eval and Print is done like this:

```
object evalPrint extends ExpMerge[IEval,IPrint] {
  val alg1 = eval
  val alg2 = print

  def lift(x: IEval, y: IPrint) = new IEval with IPrint {
    def eval = x.eval
    def print = y.print
  }
}
```

With the composed algebra EvalPrint, the expression can be constructed only once:

```
val e = exp(evalPrint)
println(e.print + "=" + e.eval)
```

Unfortunately, the composition is still done in a cubersome way. Specialized combinators are needed for each Object Algebra interface and a lot of boilerplate code for each composition. Workarounds are further proposed such as using generic combinators and reflection [Oliveira et al., 2013] or a combination of macros and implicits [Rendel et al., 2014].

Dependent operations The second issue is that *it is hard to model dependent operations*. With conventional Object Algebras, the dependent operation has to be defined together with what it depends on. Recall the pretty-printer that depends on the evaluation discussed in Section 4.1. It is defined as follows:

```
trait DPrint extends ExpAlg[IEval with IPrint] {
  def Lit = n => new IEval with IPrint {
    def eval = n
    def print = n.toString
  }
  def Add = (e1,e2) => new IEval with IPrint {
    def eval = e1.eval + e2.eval
    def print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")"
  }
}
```

The type parameter Exp is instantiated as IEval **with** IPrint for allowing both eval and print invocations on expressions. Such an implementation is non-modular because the implementation of Eval is repeated inside DPrint and DPrint is tightly coupled with a particular implementation of evaluation.

Generalized Object Algebras To account for dependencies modularly, a generalization of Object Algebras has been proposed by Oliveira et al. [2013]. The expression Object Algebra interface is generalized by distinguishing negative (input) and positive (output) occurrences of the expression type. For example, ExpAlg can be generalized in the following way:

```
trait GExpAlg[Exp,OExp] {
  def Lit: Int        => OExp
  def Add: (Exp,Exp) => OExp
}
```

where an additional type parameter OExp is introduced for capturing positive (output) occurrences of the expression type. Here, the return type positions of the two factory methods are positive and hence are replaced by OExp. The ordinary Object Algebra interface can be restored by making Exp and OExp consistent, which is convenient for defining algebras without dependencies:

```
type ExpAlg[Exp] = GExpAlg[Exp,Exp]
```

By doing this, evaluation and printing algebras can be defined as before. Furthermore, dependent printing can be modularly defined with generalized Object Algebras:

```
trait DPrint extends GExpAlg[IEval with IPrint,IPrint] {
  def Lit = n => new IPrint {
    def print = n.toString
  }
  def Add = (e1,e2) => new IPrint {
    def print = if (e1.eval == 0) e2.print
    else "(" ++ e1.print ++ "+" ++ ")"
  }
}
```

The dependency on evaluation is expressed by instantiating the input type as IEval **with** IPrint and the output type as IPrint. The dependency is modular because DPrint does not depend on a particular implementation of evaluation. The dependency can be fulfilled by composing DPrint with any other algebra that implements ExpAlg[IEval] such as Eval. However this requires a generalized Object Algebra combinator for performing such composition.

Summary The EP illustrates some fundamental difficulties of writing modular code in current programming languages. Techniques such as Object Algebras provide a solution for such problems, but they have their own limitations. These limitations partly arise from the lack of programming language support. In particular:

- **Unconventional programming style:** The programming style required to program with Object Algebras is quite unconventional compared to standard OOP code. For instance, since constructors are avoided, all objects must be constructed relative to an Object Algebra (or factory), similarly to the method `exp`. Moreover, the code required for programming with Object Algebras is somewhat verbose.

- **No built-in composition:** Scala (or other OOP languages) have built-in support for a form of inheritance, which provides a mechanism to compose code. However, such OOP languages do not support the composition of Object Algebras. Composing Object Algebras in such languages is possible, but requires the explicit definition of composition operators, which have to be defined for each Object Algebra interface.

- **No built-in support for modular dependencies:** Dependencies are quite common in programming. All realistic software will involve multiple forms of dependencies. However, using simple Object Algebras forces dependencies to be written in a non-modular way. Writing modular dependencies is possible with a generalization of Object Algebra interfaces and specialized composition operators. However, such composition operators require a lot of boilerplate code, generalized Object Algebras require a careful manual encoding that distinguishes positive and negative occurrences, and the programming style involved in such code is generally quite heavyweight and unconventional.

2.5 Disjoint Intersection Types

Intersection types are useful to model modular programs, as we have seen in the previous section. In particular, the Object Algebra combinators use intersection types. However, a missing feature in Scala for programming with intersection types is a merge operator. Without this operator, it is sometimes necessary to simulate a merge operator in Scala using meta-programming or reflection [Oliveira et al., 2013; Rendel et al., 2014], which results in convoluted error messages, performance penalties and, more generally, lack of modular type-checking.

Recent developments on disjoint intersection types [Oliveira et al., 2016] provide an alternative approach that supports a native merge operator. The λ_i calculus [Oliveira et al., 2016] was the first calculus with disjoint intersection types and addressed the incoherence problem of intersection types with a merge operator [Dunfield, 2014] by introducing the notion of disjointness. The F_i calculus [Alpuim et al., 2017] extends λ_i with a form of parametric polymorphism called *disjoint polymorphism*, where type parameters can be constrained to be disjoint with a specific type. On the other hand, λ_i^+ [Bi et al., 2018] extends λ_i with BCD-style distributive subtyping, enabling nested composition. The F_i^+ calculus [Bi et al., 2019] combines disjoint intersection types, disjoint polymorphism, and nested composition, enabling all the foundational ingredients for Compositional Programming.

We review F_i^+ only because it is the most comprehensive calculus on disjoint intersection types.

$$
\begin{array}{llcl}
\text{Types} & \tau & ::= & \mathbf{Int} \mid \alpha \mid \top \mid \bot \mid \tau_1 \rightarrow \tau_2 \mid \forall(\alpha * \tau_1).\tau_2 \mid \tau_1 \,\&\, \tau_2 \mid \{l\!:\!\tau\} \\
\text{Expressions} & e & ::= & i \mid x \mid \top \mid \lambda\!\!\!\lambda x.e \mid e_1\ e_2 \mid \Lambda(\alpha * \tau).e \mid e\ \tau \mid e_1 \,,,\, e_2 \mid \{l = e\} \mid e.l \\
 & & \mid & \mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2 \\
\text{Term contexts} & \Gamma & ::= & \bullet \mid \Gamma, x : \tau \\
\text{Type contexts} & \Delta & ::= & \bullet \mid \Delta, \alpha * \tau
\end{array}
$$

Figure 2.1: F_i^+ *syntax extened with (recursive) let bidings.*

$\boxed{\tau_1 <: \tau_2}$

$$
\textsc{TS-refl}\ \ \tau <: \tau
\qquad
\textsc{TS-trans}\ \ \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}
\qquad
\textsc{TS-top}\ \ \tau <: \top
\qquad
\textsc{TS-bot}\ \ \bot <: \tau
\qquad
\textsc{TS-rcd}\ \ \frac{\tau_1 <: \tau_2}{\{l\!:\!\tau_1\} <: \{l\!:\!\tau_2\}}
\qquad
\textsc{TS-andL}\ \ \tau_1 \,\&\, \tau_2 <: \tau_1
$$

$$
\textsc{TS-andR}\ \ \tau_1 \,\&\, \tau_2 <: \tau_2
\qquad
\textsc{TS-and}\ \ \frac{\tau_1 <: \tau_2 \qquad \tau_1 <: \tau_3}{\tau_1 <: \tau_2 \,\&\, \tau_3}
\qquad
\textsc{TS-arr}\ \ \frac{\tau_3 <: \tau_1 \qquad \tau_2 <: \tau_4}{\tau_1 \rightarrow \tau_2 <: \tau_3 \rightarrow \tau_4}
\qquad
\textsc{TS-topArr}\ \ \top <: \top \rightarrow \top
\qquad
\textsc{TS-topRcd}\ \ \top <: \{l\!:\!\top\}
$$

$$
\textsc{TS-topAll}\ \ \top <: \forall(\alpha * \top).\top
\qquad
\textsc{TS-forall}\ \ \frac{\tau_2 <: \tau_4 \qquad \tau_3 <: \tau_1}{\forall(\alpha * \tau_1).\tau_2 <: \forall(\alpha * \tau_3).\tau_4}
\qquad
\textsc{TS-distArr}\ \ (\tau_1 \rightarrow \tau_2) \,\&\, (\tau_1 \rightarrow \tau_3) <: \tau_1 \rightarrow (\tau_2 \,\&\, \tau_3)
$$

$$
\textsc{TS-distRcd}\ \ \{l\!:\!\tau_1\} \,\&\, \{l\!:\!\tau_2\} <: \{l\!:\!\tau_1 \,\&\, \tau_2\}
\qquad
\textsc{TS-distAll}\ \ \forall(\alpha * \tau_1).\tau_2 \,\&\, \forall(\alpha * \tau_1).\tau_3 <: \forall(\alpha * \tau_1).\tau_2 \,\&\, \tau_3
$$

Figure 2.2: F_i^+ *subtyping.*

Syntax Figure 2.1 gives the syntax of F_i^+. Metavariable τ ranges over types. Types include integers **Int**, type variables α, the top type $\top$, the bottom type $\bot$, arrows $\tau_1 \rightarrow \tau_2$, disjoint quantification $\forall(\alpha * \tau_1).\tau_2$, intersections $\tau_1 \,\&\, \tau_2$, and single-field record types $\{l : \tau\}$. Metavariable e ranges over expressions. Expressions include integer literals i, term variables x, the top value $\top$, lambda abstractions $\lambda\!\!\!\lambda x.e$, term applications $e_1\ e_2$, type abstractions $\Lambda(\alpha * \tau).e$ with α constrained to be disjoint with τ, type applications $e\ \tau$, merges $e_1 \,,,\, e_2$, single-field records $\{l = e\}$, record projections $e.l$ and (recursive) let expressions $\mathbf{let}\ x : \tau = e_1\ \mathbf{in}\ e_2$.

Subtyping Figure 2.2 shows the subtyping rules of the form $\tau_1 <: \tau_2$. The subtyping relation is reflective (TS-REFL) and transitive (S-TRANS). Rules for top types (TS-TOP), bottom types (TS-BOT), function types (TS-ARR) and record types (TS-RCD) are standard. The three rules on intersection types (TS-ANDL, TS-ANDR and TS-AND) state that $\tau_1 \,\&\, \tau_2$ is the greatest lower bound for τ_1 and τ_2. The BCD-style distributive rules [Barendregt et al., 1983] (TS-DISTARR, TS-DISTRCD and TS-DISTALL) are particularly interesting, since they enable nested composition.

Typing Figure 2.3 gives the bidirectional type system employed by F_i^+: under the contexts Δ and Γ, the inference mode ($\Rightarrow$) synthesizes a type τ from an expression e while the

$$\boxed{\Delta; \Gamma \vdash e \Rightarrow \tau}$$

$$\textsc{TT-top}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash \top \Rightarrow \top} \qquad \textsc{TT-nat}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma}{\Delta; \Gamma \vdash i \Rightarrow \mathbf{Int}} \qquad \textsc{TT-var}\ \frac{\vdash \Delta \qquad \Delta \vdash \Gamma \qquad (x : A) \in \Gamma}{\Delta; \Gamma \vdash x \Rightarrow A}$$

$$\textsc{TT-app}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 \Leftarrow \tau_1}{\Delta; \Gamma \vdash e_1\ e_2 \Rightarrow \tau_2} \qquad \textsc{TT-merge}\ \frac{\Delta; \Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Delta; \Gamma \vdash e_2 \Rightarrow \tau_2 \qquad \Delta \vdash \tau_1 * \tau_2}{\Delta; \Gamma \vdash e_1 \,,,\, e_2 \Rightarrow \tau_1 \,\&\, \tau_2}$$

$$\textsc{TT-anno}\ \frac{\Delta; \Gamma \vdash e \Leftarrow \tau}{\Delta; \Gamma \vdash e : \tau \Rightarrow \tau} \qquad \textsc{TT-rcd}\ \frac{\Delta; \Gamma \vdash e \Rightarrow \tau}{\Delta; \Gamma \vdash \{l = e\} \Rightarrow \{l : \tau\}} \qquad \textsc{TT-proj}\ \frac{\Delta; \Gamma \vdash e \Rightarrow \{l : \tau\}}{\Delta; \Gamma \vdash e.l \Rightarrow \tau}$$

$$\textsc{TT-tabs}\ \frac{\Delta, \alpha * \tau_1; \Gamma \vdash e \Rightarrow \tau_2}{\Delta; \Gamma \vdash \Lambda(\alpha * \tau_1).e \Rightarrow \forall(\alpha * \tau_1).\tau_2} \qquad \textsc{TT-tapp}\ \frac{\Delta; \Gamma \vdash e \Rightarrow \forall(\alpha * \tau_2).\tau_3 \qquad \Delta \vdash \tau_1 * \tau_2}{\Delta; \Gamma \vdash e\ \tau_1 \Rightarrow [\tau_1/\alpha]\tau_3}$$

$$\textsc{TT-let}\ \frac{\Delta; \Gamma, x : A \vdash e_1 \Leftarrow A \qquad \Delta; \Gamma, x : A \vdash e_2 \Rightarrow B}{\Delta; \Gamma \vdash \mathbf{let}\ x : A = e_1\ \mathbf{in}\ e_2 \Rightarrow B}$$

$$\boxed{\Delta; \Gamma \vdash e \Leftarrow \tau}$$

$$\textsc{TT-abs}\ \frac{\Delta \vdash \tau_1 \qquad \Delta; \Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Delta; \Gamma \vdash \lambda x.e \Leftarrow \tau_1 \to \tau_2} \qquad \textsc{TT-sub}\ \frac{\Delta; \Gamma \vdash e \Rightarrow \tau_2 \qquad \tau_2 <: \tau_1}{\Delta; \Gamma \vdash e \Leftarrow \tau_1}$$

Figure 2.3: F_i^+ *typing rules.*

checking mode ($\Leftarrow$) checks the type of an expression against τ. $\vdash \Delta$ and $\Delta \vdash \Gamma$ ensure the well-formedness of contexts. The rule TT-MERGE infers the merge of two expressions as an intersection type if their types are *disjoint*. The rule TT-TAPP additionally checks that the supplied type τ_1 is disjoint with the constraining type τ_2.

Disjointness The disjointness judgement ($\Delta \vdash \tau_1 * \tau_2$), shown in Figure 2.4, ensures that the merge of τ_1 and τ_2 is conflict-free. The disjointness judgement further relies on the definition of top-like types ($\rceil\tau\lceil$) and a disjoint axiom ($\tau_1 *_{ax} \tau_2$). Top-like types are types isomorphic to $\top$, which was a concept firstly introduced by Barendregt et al. [1983] and then employed by F_i^+ and its precursors [Oliveira et al., 2016; Alpuim et al., 2017; Bi et al., 2019] for proving coherence. An important property of top-like types is that they are disjoint to any other types Alpuim et al. [2017]. Furthermore, the definition of top-like types is crucial for defining disjointness and the inclusion of types such as **Int** → $\top$ in the class of top-like types, which is important to ensure the disjointness of function types and in turn enables merges with multiple functions. A more detailed discussion can be found in work by Huang and Oliveira [2020].

$\boxed{\Delta \vdash \tau_1 * \tau_2}$

$$\textsc{TD-TopL}\ \frac{\rceil \tau_1 \lceil}{\Delta \vdash \tau_1 * \tau_2} \qquad \textsc{TD-TopR}\ \frac{\rceil \tau_2 \lceil}{\Delta \vdash \tau_1 * \tau_2} \qquad \textsc{TD-Arr}\ \frac{\Delta \vdash \tau_2 * \tau_4}{\Delta \vdash \tau_1 \to \tau_2 * \tau_3 \to \tau_4} \qquad \textsc{TD-AndL}\ \frac{\Delta \vdash \tau_1 * \tau_3 \qquad \Delta \vdash \tau_2 * \tau_3}{\Delta \vdash \tau_1 \,\&\, \tau_2 * \tau_3}$$

$$\textsc{TD-AndR}\ \frac{\Delta \vdash \tau_1 * \tau_2 \qquad \Delta \vdash \tau_1 * \tau_3}{\Delta \vdash \tau_1 * \tau_2 \,\&\, \tau_3} \qquad \textsc{TD-RcdEq}\ \frac{\Delta \vdash \tau_1 * \tau_2}{\Delta \vdash \{l : \tau_1\} * \{l : \tau_2\}} \qquad \textsc{TD-RcdNeq}\ \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : \tau_1\} * \{l_2 : \tau_2\}}$$

$$\textsc{TD-TVarL}\ \frac{(\alpha * \tau_1) \in \Delta \qquad \tau_1 <: \tau_2}{\Delta \vdash \alpha * \tau_2} \qquad \textsc{TD-TVarR}\ \frac{(\alpha * \tau_1) \in \Delta \qquad \tau_1 <: \tau_2}{\Delta \vdash \tau_2 * \alpha} \qquad \textsc{TD-Forall}\ \frac{\Delta, \alpha * \tau_1 \,\&\, \tau_3 \vdash \tau_2 * \tau_4}{\Delta \vdash \forall(\alpha * \tau_1).\tau_2 * \forall(\alpha * \tau_3).\tau_4}$$

$$\textsc{TD-Ax}\ \frac{\tau_1 *_{ax} \tau_2}{\Delta \vdash \tau_1 * \tau_2}$$

$\boxed{\rceil \tau \lceil}$

$$\textsc{TL-Top}\ \rceil \top \lceil \qquad \textsc{TL-And}\ \frac{\rceil \tau_1 \lceil \qquad \rceil \tau_2 \lceil}{\rceil \tau_1 \,\&\, \tau_2 \lceil} \qquad \textsc{TL-Arr}\ \frac{\rceil \tau_2 \lceil}{\rceil \tau_1 \to \tau_2 \lceil} \qquad \textsc{TL-Rcd}\ \frac{\rceil \tau \lceil}{\rceil \{l : \tau\} \lceil} \qquad \textsc{TL-All}\ \frac{\rceil \tau_2 \lceil}{\rceil \forall(\alpha * \tau_1).\tau_2 \lceil}$$

Figure 2.4: F_i^+ *disjointness.*

Semantics The semantics of F_i^+ is given by elaborating to F_{co}, a variant of System F extended with products and explicit coercions. Details of F_{co} are beyond the scope of this thesis. We refer interested readers to the original F_i^+ paper [Bi et al., 2019].

2.6 SEDEL and First-Class Traits

As a core calculus, F_i^+ lacks higher-level programming abstractions to make programming convenient. SEDEL [Bi and Oliveira, 2018] is a surface language built upon F_i that supports *first-class* traits. That is, unlike Scala traits, SEDEL traits are expressions that can be passed to a function, assigned to a variable or returned as values. Furthermore, they can be composed to achieve a form of multiple inheritance. Such composition is enabled by the merge operator of F_i.

Trait expressions and self-types Like Scala traits, SEDEL traits can be annotated with self-types for expressing the dependency on some methods that are implemented by other traits. For example, we can have two mutually dependent traits that respectively implement the methods for testing whether a number is even or odd:

```
type Even = { isEven : Int -> Bool };
type Odd  = { isOdd  : Int -> Bool };

even = trait [self: Odd] => {
  isEven (n : Int) = if n == 0 then true else self.isOdd (n - 1)
} : Even;
odd = trait [self: Even] => {
  isOdd (n : Int) = if n == 0 then false else self.isEven (n - 1)
} : Odd;
```

`Even` and `Odd` are type aliases respectively bound to the record types declaring `isEven` and `isOdd`. By annotating the self-type as `[self: Odd]`, `even` can call `isOdd` via `self` for implementing `isEven` in its body. The `Even` annotation in the end of the trait expression makes sure that `isEven` has been implemented.

Trait types In the previous example, the type of `even` is `Trait[Odd,Even]` and the type of `odd` is `Trait[Even,Odd]`. Trait types in SEDEL distinguish between the *required* and the *provided* interface. The required interface describes the methods that the trait needs for providing its functionality, playing a similar role to *abstract methods* in other OOP languages. Meanwhile, the provided interface describes the functionality that the trait offers. For the case of `Trait[Odd,Even]`, `Odd` is the required interface while `Even` is the provided interface. When nothing is required, we can just write `Trait[A]` instead of `Trait[Top,A]`.

Trait instantiations A trait can be instantiated into an object (a record) using a `new` expression only when its required interface is met. For example, the following attempt to instantiate `even` fails:

```
new[Even] even -- Type Error!
```

Here the object's type, `Even`, must be explicitly specified inside `[]` of the `new` expression. The above instantiation fails because the required interface of `even`, `Odd`, is neither implemented by the trait `even` nor its parents. Nevertheless, `even` can be instantiated together with `odd` since the two traits implement each other's required interface:

```
evenOdd = new[Even & Odd] even & odd; -- OK!
main = evenOdd.isEven 2               --> true
```

The two traits are instantiated into a single object of type `Even & Odd` that implements both `isEven` and `isOdd` methods.

First-class traits, disjoint polymorphism, and dynamic inheritance An example that differentiates SEDEL's traits from Scala's traits is:

```
combine A [B * A] (x : Trait[A]) (y : Trait[B]) = trait inherits x & y => {};
```

The function `combine` takes two traits, `x` and `y`, and returns a trait that inherits both `x` and `y`. The definition of `combine` illustrates three key features of SEDEL: *disjoint polymorphism*, *first-class traits* and *dynamic inheritance*. Firstly, `combine` is a polymorphic function and the notation `[B * A]`, means that the type parameter `B` is *disjoint* with `A`. This constraint ensures that inheriting `x` and `y` simultaneously will have no conflicts. Secondly, traits are passed as arguments (`x` and `y`) and a trait is the return value of `combine`, showing that traits are *first-class* values. Thirdly, note that what the `trait` expression inherits (`x` and `y`) are parameters of `combine`, which are statically unknown. Such kind of *dynamic* *inheritance* is not possible in conventional statically typed OOP languages (like Scala or Java), where classes must be statically known, and they cannot be passed as arguments.

Resolving conflicts Trait composition (or inheritance) in SEDEL follows the traditional trait model [Schärli et al., 2003] where two traits cannot have conflicts for composition to be successful. A benefit of the trait model is that composition is *commutative* and *associative*, which ensures that the order of composition is irrelevant. In the implementation of SEDEL, trait composition is encoded in terms of the merge operator, which is itself *commutative* and *associative* and does not allow for overlapping (or conflicting) values [Bi et al., 2018]. In the case that two traits being composed have conflicts, those conflicts must be explicitly resolved in SEDEL. Suppose that we have two traits `t1` and `t2` that contain conflicting fields:

```
t1 = trait => { f = 1; g = "a" };
t2 = trait => { f = 2; g = "b" };
```

For a trait that would like to inherit from both `t1` and `t2`, the conflicts must be explicitly resolved. Otherwise, a type error will be reported saying that `t1` and `t2` are not disjoint. One way to resolve the conflicts is:

```
t3 = trait [self: Top] inherits t1 \ {f: Int} & t2 \ {g: String} => {
  override f = super.f + (t1 ^ self).f
};
```

The conflicts are resolved by using the *exclusion* operator (\) to remove `f` and `g` respectively from `t1` and `t2`. Together with a self-type annotation `[self: Top]`, the excluded methods from parents can still be accessed via the *forwarding* operator (^) [Bi and Oliveira, 2018]. Here, the overridden `f` sums up the inherited `f` from `t2` via **`super`** and the excluded `f` from `t1` via the forwarding expression. We can construct an object from `t3` to test that `f` actually returns 3:

```
main = (new[{f: Int; g: String}] t3).f  --> 3
```

Chapter 3

Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality

In this chapter, we focus on using plain design patterns to enhance the modularity of programs. We first connect shallow embeddings to OOP and argue that OOP mechanisms improve the modularity of shallow embeddings. We make our argument by employing the Extensible Interpreter pattern and Object Algebras in modularizing several embedded domain-specific languages (EDSLs).

3.1 Introduction

Since Hudak's seminal paper [Hudak, 1998] on EDSLs, existing languages have been used to directly encode DSLs. Two common approaches to EDSLs are the so-called *shallow* and *deep* embeddings. Deep embeddings emphasize a *syntax*-first approach: the abstract syntax is defined first using a data type, and then interpretations of the abstract syntax follow. The role of interpretations in deep embeddings is to map syntactic values into semantic values in a semantic domain. Shallow embeddings emphasize a *semantics*-first approach, where a semantic domain is defined first. In the shallow approach, the operations of the EDSLs are interpreted directly into the semantic domain. Therefore there is no data type representing uninterpreted abstract syntax.

The trade-offs between shallow and deep embeddings have been widely discussed [Svenningsson and Axelsson, 2012; Jovanovic et al., 2014]. Deep embeddings enable transformations on the abstract syntax tree (AST), and multiple interpretations are easy to implement. Shallow embeddings enforce the property of *compositionality* by construction and are easily extended with new EDSL syntax. Such discussions lead to a generally accepted belief that it is hard to support multiple interpretations [Svenningsson and Axelsson, 2012] and AST transformations in shallow embeddings.

Compositionality is considered a sign of good language design, and it is one of the hallmarks of denotational semantics. Compositionality means that a denotation (or interpretation) of a language is constructed from the denotation of its parts. Compositionality leads to a modular semantics, where adding new language constructs does not require changes in the semantics of existing constructs. Because compositionality offers a guideline for good language design, Erwig and Walkingshaw [2012] argue that a semantics-first approach to EDSLs is superior to a syntax-first approach. Shallow embeddings fit well with such a semantics-driven approach. Nevertheless, the limitations of shallow embeddings compared to deep embeddings can deter their use.

This chapter shows that, given adequate language support, having multiple modular interpretations in shallow DSLs is not only possible but simple. Therefore we aim to debunk the belief that multiple interpretations are hard to model with shallow embeddings. Several previous authors [Gibbons and Wu, 2014; Erwig and Walkingshaw, 2012] already observed that, by using products and projections, multiple interpretations can be supported with a cumbersome and often non-modular encoding. Moreover, it is also known that multiple interpretations *without dependencies* on other interpretations are modularized easily using variants of Church encodings [Gibbons and Wu, 2014; Carette et al., 2009; Oliveira and Cook, 2012]. We show that a solution for multiple interpretations, including dependencies, is encodable naturally when the host language combines functional features with common OO features, such as *subtyping*, *inheritance*, and *type-refinement*.

At the center of this chapter is Reynolds' [1978] idea of *procedural abstraction*, which enables us to relate shallow embeddings and OOP directly. With procedural abstraction, data is characterized by the operations that are performed over it. This chapter builds on two independently observed connections to procedural abstraction:

$$\begin{tikzcd}\textit{Shallow Embeddings} \arrow[r, leftrightarrow, "\text{Gibbons and Wu [2014]}"] & \textit{Procedural Abstraction} \arrow[r, leftrightarrow, "\text{Cook [2009]}"] & \textit{OOP}\end{tikzcd}$$

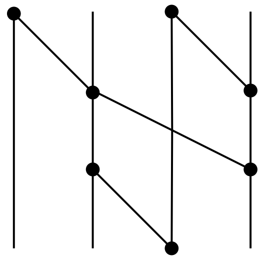
The first connection is between procedural abstraction and shallow embeddings. As Gibbons and Wu [2014] state, “*it was probably known to Reynolds, who contrasted deep embeddings (user defined types) and shallow (procedural data structures)*”. Gibbons and Wu noted the connection between shallow embeddings and procedural abstractions, although they did not go into much detail. The second connection is between OOP and procedural abstraction, which was discussed in depth by Cook [2009].

We make our arguments concrete using Gibbons and Wu's [2014] examples, where procedural abstraction is used in Haskell to model a simple *shallow* EDSL. We recode that EDSL in Scala using Wang and Oliveira's [2016] extensible interpreter pattern. The resulting Scala version has *modularity* advantages over the Haskell version, due to the use of subtyping, inheritance, and type-refinement. In particular, the Scala code can easily express modular interpretations that may *not only depend on themselves but also depend on other modular interpretations*, leading to our motto: *beyond simple compositionality*.

While Haskell does not natively support subtyping, inheritance, and type-refinement, its powerful and expressive type system is sufficient to encode similar features. Therefore we can port back to Haskell some of the ideas used in the Scala solution using an improved Haskell encoding that has similar (and sometimes even better) benefits in terms of modularity. In essence, in the Haskell solution we encode a form of subtyping on pairs using type classes. This is useful to avoid explicit projections, that clutter the original Haskell solution. Inheritance is encoded by explicitly delegating interpretations using Haskell superclasses. Finally, type-refinement is simulated using the subtyping typeclass to introduce subtyping constraints.

```
⟨circuit⟩ ::= id ⟨positive number⟩
            | fan ⟨positive number⟩
            | ⟨circuit⟩ beside ⟨circuit⟩
            | ⟨circuit⟩ above ⟨circuit⟩
            | stretch ⟨positive numbers⟩ ⟨circuit⟩
            | ( ⟨circuit⟩ )
```

Figure 3.1: *The grammar of* Scans.



```
(fan 2 beside fan 2) above
(stretch 2 2 fan 2) above
(id 1 beside fan 2 beside id 1)
```

Figure 3.2: *The Brent-Kung circuit of width 4.*

While the techniques are still cumbersome for AST transformations, yielding efficient shallow EDSLs is still possible via staging [Rompf and Odersky, 2010; Carette et al., 2009]. By removing the limitation of multiple interpretations, we enlarge the applicability of shallow embeddings. A concrete example is our case study, which refactors an external SQL query processor that employs deep embedding techniques [Rompf and Amin, 2015] into a shallow EDSL. The refactored implementation allows both new (possibly dependent) interpretations and new constructs to be introduced modularly without sacrificing the performance.

The complete code for all examples and case study is available at:

https://github.com/wxzh/shallow-dsl

3.2 Shallow Object-Oriented Programming

This section shows how OOP and shallow embeddings are related via procedural abstraction. We use the same DSL presented by Gibbons and Wu [2014] as a running example. We first give the original shallow embedded implementation in Haskell, and rewrite it towards an "OOP style". Then translating the program into a functional OOP language like Scala becomes straightforward.

3.2.1 Scans: A DSL for Parallel Prefix Circuits

Scans [Hinze, 2004] is a DSL for describing parallel prefix circuits. Given an associative binary operator $\bullet$, the prefix sum of a non-empty sequence $x_1, x_2, ..., x_n$ is $x_1, x_1 \bullet x_2, ..., x_1 \bullet x_2 \bullet ... \bullet x_n$. Such computation can be performed in parallel for a parallel prefix circuit. Parallel prefix circuits have many applications, including binary addition and sorting algorithms. The grammar of Scans is given in Figure 3.1. Scans has five constructs: two primitives (*id* and *fan*) and three combinators (*beside*, *above* and *stretch*). Their meanings are: *id n* contains *n* parallel wires; *fan n* has *n* parallel wires with the leftmost wire connected to all other wires from top to bottom; c_1 *beside* c_2 joins two circuits c_1 and c_2 horizontally; c_1 *above* c_2 combines two circuits of the same width vertically; *stretch ns c* inserts wires into the circuit *c* so that the i^{th} wire of *c* is stretched to a position of $ns_1 + \ldots + ns_i$, resulting in a new circuit of width by summing up *ns*. Figure 3.2 visualizes a circuit constructed using all these five constructs. The structure of this circuit is explained as follows. The whole circuit is vertically composed by three sub-circuits: the top sub-circuit is a two 2-*fan*s put side by side; the middle sub-circuit is a 2-*fan* stretched by inserting a wire on the left-hand side of its first and second wire; the bottom sub-circuit is a 2-*fan* in the middle of two 1-*id*s.

3.2.2 Shallow Embeddings and OOP

Shallow embeddings define a language directly by encoding its semantics using procedural abstraction. In the case of Scans, a shallow embedded implementation (in Haskell) conforms to the following types:

```
type Circuit = ...    -- the operations we wish to support for circuits
id           :: Int → Circuit
fan          :: Int → Circuit
beside       :: Circuit → Circuit → Circuit
above        :: Circuit → Circuit → Circuit
stretch      :: [Int] → Circuit → Circuit
```

The type *Circuit*, representing the semantic domain, is to be filled with a concrete type according to the semantics. Each construct is declared as a function that produces a *Circuit*. Suppose that the semantics of Scans calculates the width of a circuit. The definitions are:

```
type Circuit = Int
id n         = n
fan n        = n
beside c1 c2 = c1 + c2
above c1 c2  = c1
stretch ns c = sum ns
```

For this interpretation, the Haskell domain is simply *Int*. This means that we will get the width immediately after the construction of a circuit. Note that the *Int* domain for *width* is a degenerate case of procedural abstraction: *Int* can be viewed as a no argument function. In Haskell, due to laziness, *Int* is a good representation. In a call-by-value language, a no-argument function () → *Int* is more appropriate to deal correctly with potential control-flow language constructs.

Now we are able to construct the circuit in Figure 3.2 using these definitions:

```
// object interface
trait Circuit1 {def width : Int}
// concrete implementations
trait Id1 extends Circuit1 {
  val n : Int
  def width = n
}
trait Fan1 extends Circuit1 {
  val n : Int
  def width = n
}
```

```
trait Beside1 extends Circuit1 {
  val c1, c2 : Circuit1
  def width = c1.width + c2.width
}
trait Above1 extends Circuit1 {
  val c1, c2 : Circuit1
  def width = c1.width
}
trait Stretch1 extends Circuit1 {
  val ns : List [Int]; val c : Circuit1
  def width = ns.sum
}
```

Figure 3.3: *Circuit interpretation in Scala.*

```
> (fan 2 ‘beside‘ fan 2) ‘above‘
| stretch [2, 2] (fan 2) ‘above‘
| (id 1 ‘beside‘ fan 2 ‘beside‘ id 1)
4
```

Towards OOP An *isomorphic encoding* of *width* is given below, where a record with one field captures the domain and is declared as a **newtype**:

```
newtype Circuit1 = Circuit1 {width1 :: Int}
id1 n            = Circuit1 {width1 = n}
fan1 n           = Circuit1 {width1 = n}
beside1 c1 c2    = Circuit1 {width1 = width1 c1 + width1 c2}
above1 c1 c2     = Circuit1 {width1 = width1 c1}
stretch1 ns c    = Circuit1 {width1 = sum ns}
```

The implementation is still shallow because Haskell's **newtype** does not add any operational behavior to the program. Hence the two programs are effectively the same. However, having fields makes the program look more like an OO program.

Porting to Scala Indeed, we can easily translate the program from Haskell to Scala, as shown in Figure 3.3. The idea is to map Haskell's record types into an object interface (modeled as a **trait** in Scala) $Circuit_1$, and Haskell's field declarations become method declarations. Object interfaces make the connection to procedural abstraction clear: data is modeled by the operations that can be performed over it. Each case in the semantic function corresponds to a concrete implementation of $Circuit_1$, where function parameters are captured as immutable fields.

This implementation is essentially how we would model SCANS with an OOP language in the first place. A minor difference is the use of traits instead of classes in implementing $Circuit_1$. Although a class definition like

```
class Id1 (n : Int) extends Circuit1 {def width = n}
```

is more common, some modularity offered by the trait version (e.g. mixin composition) is lost. To use this Scala implementation in a manner similar to the Haskell implementation, we need some smart constructors for creating objects conveniently:

```
def id (x : Int)                         = new Id1      {val n = x}
def fan (x : Int)                        = new Fan1     {val n = x}
def beside (x : Circuit1, y : Circuit1)  = new Beside1  {val c1 = x; val c2 = y}
def above (x : Circuit1, y : Circuit1)   = new Above1   {val c1 = x; val c2 = y}
def stretch (x : Circuit1, xs : Int*)    = new Stretch1 {val ns = xs.toList; val c = x}
```

Now we are able to construct the circuit shown in Figure 3.2 in Scala:

```
val circuit = above (beside (fan (2), fan (2)),
                     above (stretch (fan (2), 2, 2),
                            beside (beside (id (1), fan (2)), id (1))))
```

Finally, calling *circuit.width* will return 4 as expected.

As this example illustrates, shallow embeddings and straightforward OO programming are closely related. The syntax of the Scala code is not as concise as the Haskell version due to some extra verbosity caused by trait declarations and smart constructors. Nevertheless, the code is still quite compact and elegant, and the Scala implementation has advantages in terms of modularity, as we shall see next.

3.3 Multiple Interpretations in Shallow Embeddings

An often stated limitation of shallow embeddings is that multiple interpretations are difficult. Gibbons and Wu [2014] work around this problem by using tuples. However, their encoding needs to modify the original code and thus is non-modular. This section illustrates how various types of interpretations can be *modularly* defined using standard OOP mechanisms, and compares the result with Gibbons and Wu's Haskell implementations.

3.3.1 Simple Multiple Interpretations

A single interpretation may not be enough for realistic DSLs. For example, besides *width*, we may want to have another interpretation that calculates the depth of a circuit in Scans.

Multiple interpretations in Haskell Here is Gibbons and Wu's [2014] solution:

```
type Circuit2 = (Int, Int)
id2 n         = (n, 0)
fan2 n        = (n, 1)
above2 c1 c2  = (width c1, depth c1 + depth c2)
```

```
beside2 c1 c2 = (width c1 + width c2, depth c1 `max` depth c2)
stretch2 ns c  = (sum ns, depth c)

width = fst
depth = snd
```

A tuple is used to accommodate multiple interpretations, and each interpretation is defined as a projection on the tuple. However, this solution is not modular because it relies on defining the two interpretations (*width* and *depth*) simultaneously. It is not possible to reuse the independently defined *width* interpretation in Section 3.2.2. Whenever a new interpretation is needed (e.g. *depth*), the original code has to be revised: the arity of the tuple must be incremented and the new interpretation has to be appended to each case.

Multiple interpretations in Scala In contrast, a Scala solution allows new interpretations to be introduced in a modular way:

```
trait Circuit2 extends Circuit1 {def depth : Int} // subtyping
trait Id2 extends Id1 with Circuit2 {def depth = 0}
trait Fan2 extends Fan1 with Circuit2 {def depth = 1}
trait Above2 extends Above1 with Circuit2 {      // inheritance
  override val c1, c2 : Circuit2                 // covariant type-refinement
  def depth = c1.depth + c2.depth
}
trait Beside2 extends Beside1 with Circuit2 {
  override val c1, c2 : Circuit2
  def depth = Math.max (c1.depth, c2.depth)
}
trait Stretch2 extends Stretch1 with Circuit2 {
  override val c : Circuit2
  def depth = c.depth
}
```

The encoding relies on three OOP abstraction mechanisms: *inheritance*, *subtyping*, and *type-refinement*. Specifically, *Circuit*$_2$ is a subtype of *Circuit*$_1$, which extends the semantic domain with a *depth* method. Concrete cases, for instance *Above*$_2$, implement *Circuit*$_2$ by inheriting *Above*$_1$ and implementing *depth*. Also, fields of type *Circuit*$_1$ are covariantly refined as type *Circuit*$_2$ to allow *depth* invocations. Importantly, all definitions for *width* in Section 3.2.2 are *modularly reused* here.

3.3.2 Dependent Interpretations

Dependent interpretations are a generalization of multiple interpretations. A dependent interpretation does not only depend on itself but also on other interpretations, which goes beyond simple compositional interpretations. An instance of dependent interpretation is *wellSized*, which checks whether a circuit is constructed correctly. The interpretation of *wellSized* is dependent because combinators like *above* use *width* in their definitions.

Dependent interpretations in Haskell In the Haskell solution given by Gibbons and Wu [2014], dependent interpretations are again defined with tuples in a non-modular way:

```
type Circuit3 = (Int, Bool)
id3 n         = (n, True)
fan3 n        = (n, True)
above3 c1 c2  = (width c1, wellSized c1 ∧ wellSized c2 ∧ width c1 ≡ width c2)
beside3 c1 c2 = (width c1 + width c2, wellSized c1 ∧ wellSized c2)
stretch3 ns c = (sum ns, wellSized c ∧ length ns ≡ width c)

wellSized = snd
```

where *width* is called in the definition of *wellSized* for $above_3$ and $stretch_3$.

Dependent interpretations in Scala Once again, it is easy to model dependent interpretation with a simple OO approach:

```
trait Circuit3 extends Circuit1 {def wellSized : Boolean} // dependency declaration
trait Id3 extends Id1 with Circuit3 {def wellSized = true}
trait Fan3 extends Fan1 with Circuit3 {def wellSized = true}
trait Above3 extends Above1 with Circuit3 {
  override val c1, c2 : Circuit3
  def wellSized =
    c1.wellSized ∧ c2.wellSized ∧ c1.width ≡ c2.width    // dependency usage
}
trait Beside3 extends Beside1 with Circuit3 {
  override val c1, c2 : Circuit3
  def wellSized = c1.wellSized ∧ c2.wellSized
}
trait Stretch3 extends Stretch1 with Circuit3 {
  override val c : Circuit3
  def wellSized = c.wellSized ∧ ns.length ≡ c.width       // dependency usage
}
```

Note that *width* and *wellSized* are defined separately. Essentially, it is sufficient to define *wellSized* while knowing only the signature of *width* in the object interface. In the definition of $Above_3$, for example, it is possible not only to call *wellSized*, but also *width*.

3.3.3 Context-sensitive Interpretations

Interpretations may rely on some context. Consider an interpretation that simplifies the representation of a circuit. A circuit can be divided horizontally into layers. Each layer can be represented as a sequence of pairs (i, j), denoting the connection from wire i to wire j. For instance, the circuit shown in Figure 3.2 has the following layout:

$$[[(0,1),(2,3)],[(1,3)],[(1,2)]]$$

The combinator *stretch* and *beside* will change the layout of a circuit. For example, if two circuits are put side by side, all the indices of the right circuit will be increased by the width of the left circuit. Hence the interpretation *layout* is also dependent, relying on itself as well as *width*. An intuitive implementation of *layout* performs these changes immediately to the affected circuit. A more efficient implementation accumulates these changes and applies them all at once. Therefore, an accumulating parameter is used to achieve this goal, which makes *layout* context-sensitive.

Context-sensitive interpretations in Haskell The following Haskell code implements (non-modular) *layout*:

```
type Circuit₄ = (Int, (Int → Int) → [[(Int, Int)]])
id₄ n         = (n, λf → [ ])
fan₄ n        = (n, λf → [[(f 0, f j) | j ← [1 .. n − 1]]])
above₄ c₁ c₂  = (width c₁, λf → layout c₁ f ++ layout c₂ f)
beside₄ c₁ c₂ = (width c₁ + width c₂,
                   λf → lzw (++) (layout c₁ f) (layout c₂ (f ∘ (width c₁+))))
stretch₄ ns c = (sum ns, λf → layout c (f ∘ pred ∘ (scanl1 (+) ns!!)))

lzw                 :: (a → a → a) → [a] → [a] → [a]
lzw f [ ] ys        = ys
lzw f xs [ ]        = xs
lzw f (x : xs) (y : ys) = f x y : lzw f xs ys

layout = snd
```

The domain of *layout* is a function type $(Int \to Int) \to [[(Int, Int)]]$, which takes a transformation on wires and produces a layout. An anonymous function is hence defined for each case, where f is the accumulating parameter. Note that f is accumulated in $beside_4$ and $stretch_4$ through function composition, propagated in $above_4$, and finally applied to wire connections in fan_4. An auxiliary definition *lzw* (stands for "long zip with") zips two lists by applying the binary operator to elements of the same index and appending the remaining elements from the longer list to the resulting list. By calling *layout* on a circuit and supplying an identity function as the initial value of the accumulating parameter, we will get the layout.

Context-sensitive interpretations in Scala Context-sensitive interpretations in Scala are unproblematic as well:

```
trait Circuit₄ extends Circuit₁ {def layout(f : Int ⇒ Int) : List[List[(Int, Int)]]}
trait Id₄ extends Id₁ with Circuit₄ {def layout(f : Int ⇒ Int) = List()}
trait Fan₄ extends Fan₁ with Circuit₄ {
  def layout(f : Int ⇒ Int) = List(for (i ← List.range(1, n)) yield (f (0), f (i)))
}
trait Above₄ extends Above₁ with Circuit₄ {
  override val c₁, c₂ : Circuit₄
```

```
    def layout (f : Int ⇒ Int) = c1.layout (f) ++ c2.layout (f)
  }
  trait Beside4 extends Beside1 with Circuit4 {
    override val c1, c2 : Circuit4
    def layout (f : Int ⇒ Int) =
      lzw (c1.layout (f), c2.layout (f.compose (c1.width + _))) (_ ++ _)
  }
  trait Stretch4 extends Stretch1 with Circuit4 {
    override val c : Circuit4
    def layout (f : Int ⇒ Int) = {
      val vs = ns.scanLeft (0) (_ + _).tail
      c.layout (f.compose (vs (_) − 1))
    }
  }
  def lzw [A] (xs : List [A], ys : List [A]) (f : (A, A) ⇒ A) : List [A] = (xs, ys) match {
    case (Nil, _)           ⇒ ys
    case (_, Nil)           ⇒ xs
    case (x :: xs, y :: ys) ⇒ f (x, y) :: lzw (xs, ys) (f)
  }
```

The Scala version captures contexts as method arguments and the implementation of *layout* is a direct translation from the Haskell version. There are some minor syntax differences that need explanations. Firstly, in Fan_4, a *for comprehension* is used for producing a list of connections. Secondly, for simplicity, anonymous functions are created without a parameter list. For example, inside $Beside_4$, $c_1.width + \_$ is a shorthand for $i \Rightarrow c_1.width + i$, where the placeholder _ plays the role of the named parameter i. Thirdly, function composition is achieved through the *compose* method defined on function values, which has a reverse composition order as opposed to $\circ$ in Haskell. Fourthly, *lzw* is implemented as a *curried function*, where the binary operator f is moved to the end as a separate parameter list for facilitating type inference.

3.3.4 An Alternative Encoding of Modular Interpretations

There is an alternative encoding of modular interpretations in Scala. For example, the *wellSized* interpretation can be re-defined like this:

```
  trait Circuit3 extends Circuit1 {def wellSized : Boolean}
  trait Id3 extends Circuit3 {
    def wellSized = true
  }
   ...
  trait Stretch3 extends Circuit3 {
    val c : Circuit3; val ns : List [Int]
    def wellSized = c.wellSized ∧ ns.length ≡ c.width
```

```
}
```

where a concrete case like Id_3 does not inherit Id_1 and leaves the *width* method unimplemented. Then, an extra step to combine *wellSized* and *width* is needed:

```
trait Id13 extends Id1 with Id3
 ...
trait Stretch13 extends Stretch1 with Stretch3
```

Compared to the previous encoding, this encoding is more modular because it decouples *wellSized* with a particular implementation of *width*. However, more boilerplate is needed for combining interpretations. Moreover, it requires some support for *multiple-inheritance*, which restricts the encoding itself from being applied to a wider range of OO languages.

3.3.5 Modular Language Constructs

Besides new interpretations, new language constructs may be needed when a DSL evolves. For example, in the case of Scans, we may want a *rstretch* (right stretch) combinator which is similar to the *stretch* combinator but stretches a circuit oppositely.

New constructs in Haskell Shallow embeddings make the addition of *rstretch* easy by defining a new function:

```
rstretch       :: [Int] → Circuit4 → Circuit4
rstretch ns c = stretch4 (1 : init ns) c `beside4` id4 (last ns − 1)
```

rstretch happens to be syntactic sugar over existing constructs. For non-sugar constructs, a new function that implements all supported interpretations is needed.

New constructs in Scala Such simplicity of adding new constructs is retained in Scala. Differently from the Haskell approach, there is a clear distinction between syntactic sugar and ordinary constructs in Scala.

In Scala, syntactic sugar is defined as a smart constructor upon other smart constructors:

```
def rstretch(ns : List[Int], c : Circuit4) = stretch(1 :: ns.init, beside(c, id(ns.last − 1)))
```

On the other hand, adding ordinary constructs is done by defining a new trait that implements $Circuit_4$. If we treated *rstretch* as an ordinary construct, its definition would be:

```
trait RStretch extends Stretch4 {
  override def layout(f : Int ⇒ Int) = {
    val vs = ns.scanLeft(ns.last − 1)(_ + _).init
    c.layout(f.compose(vs(_)))
    }
  }
```

Such an implementation of *RStretch* illustrates another strength of the Scala approach regarding modularity. Note that *RStretch* does not implement $\mathit{Circuit}_4$ directly. Instead, it inherits $\mathit{Stretch}_4$ and overrides the *layout* definition so as to reuse other interpretations as well as field declarations from $\mathit{Stretch}_4$. Inheritance and method overriding enable partial reuse of an existing language construct implementation, which is particularly useful for defining specialized constructs.

3.3.6 Discussion

Gibbons and Wu claim that in shallow embeddings new language constructs are easy to add, but new interpretations are hard. It is possible to define multiple interpretations via tuples, "*but this is still a bit clumsy: it entails revising existing code each time a new interpretation is added, and wide tuples generally lack good language support*" [Gibbons and Wu, 2014]. In other words, Haskell's approach based on tuples is essentially non-modular. However, as our Scala code shows, using OOP mechanisms both language constructs and interpretations are easy to add in shallow embeddings. Moreover, dependent interpretations are possible too, which enables interpretations that may depend on other modular interpretations and go beyond simple compositionality. The key point is that procedural abstraction combined with OOP features (subtyping, inheritance, and type-refinement) adds expressiveness over traditional procedural abstraction.

One worthy point about the Scala solution presented so far is that it is straightforward using OOP mechanisms, it uses only simple types, and dependent interpretations are not a problem. Gibbons and Wu do discuss a number of more advanced techniques [Carette et al., 2009; Swierstra, 2008] that can solve *some* of the modularity problems. In their paper, they show how to support modular *depth* and *width* (corresponding to Section 3.3.1) using the Finally Tagless [Carette et al., 2009] approach. This is possible because *depth* and *width* are non-dependent. However they do not show how to modularize *wellSized* nor *layout* (corresponding to Section 3.3.2 and 3.3.3, respectively). In Section 3.4 we revisit such Finally Tagless encoding and improve it to allow dependent interpretations, inspired by the OO solution presented in this section.

3.4 Modular interpretations in Haskell

Modular interpretations are also possible in Haskell via a variant of Church encodings that uses type classes. The original technique is due to Hinze [2006] and was shown to be modular and extensible by Oliveira et al. [2006a]. It has since been popularized under the name Finally Tagless [Carette et al., 2009] in the context of EDSLs. The idea is to use a *type class* to abstract over the signatures of constructs and define interpretations as instances of that type class. This section recodes the Scans example and compares the two modular implementations in Haskell and Scala.

3.4.1 Revisiting Scans

Here is the type class defined for Scans:

```
class Circuit c where
  id      :: Int → c
  fan     :: Int → c
  above   :: c → c → c
  beside  :: c → c → c
  stretch :: [Int] → c → c
```

The signatures are the same as what Section 3.2.2 shows except that the semantic domain is captured by a type parameter c. Interpretations such as *width* are then defined as instances of *Circuit*:

```
newtype Width = Width {width :: Int}
instance Circuit Width where
  id n          = Width n
  fan n         = Width n
  above c1 c2   = Width (width c1)
  beside c1 c2  = Width (width c1 + width c2)
  stretch ns c  = Width (sum ns)
```

where c is instantiated as a record type *Width*. Instantiating the type parameter as *Width* rather than *Int* avoids the conflict with the *depth* interpretation which also produces integers.

Multiple interpretations Adding the *depth* interpretation can now be done in a modular manner similar to *width*:

```
newtype Depth = Depth {depth :: Int}
instance Circuit Depth where
  id n          = Depth 0
  fan n         = Depth 1
  above c1 c2   = Depth (depth c1 + depth c2)
  beside c1 c2  = Depth (depth c1 `max` depth c2)
  stretch ns c  = Depth (depth c)
```

3.4.2 Modular Dependent Interpretations

Adding a modular dependent interpretation like *wellSized* is more challenging in the Finally Tagless approach. However, inspired by the OO approach we can try to mimic the OO mechanisms in Haskell to obtain similar benefits in Haskell. In what follows we explain how to encode subtyping, inheritance, and type-refinement in Haskell and how that encoding enables additional modularity benefits in Haskell.

Subtyping In the Scala solution subtyping avoids the explicit projections that are needed in the Haskell solution presented in Section 3.3. We can obtain a similar benefit in Haskell by encoding a subtyping relation on tuples in Haskell. We use the following type class, which was introduced by Bahr and Hvitved [2011], to express a subtyping relation on tuples:

```
class a ≺ b where
  prj :: a → b
instance a ≺ a where
  prj x = x
instance (a, b) ≺ a where
  prj = fst
instance (b ≺ c) ⇒ (a, b) ≺ c where
  prj = prj ∘ snd
```

In essence a type a is a subtype of a type b (expressed as $a \prec b$) if a has *the same or more* tuple components as the type b. This subtyping relation is closely related to the elaboration interpretation of *intersection types* proposed by Dunfield [2014], where Dunfield's merge operator corresponds (via elaboration) to the tuple constructor and projections are implicit and type-driven. The function *prj* simulates up-casting, which converts a value of type a to a value of type b. The three overlapping instances define the behavior of the projection function by searching for the type being projected in a compound type.

Modular *wellSized* **and encodings of inheritance and type-refinement** Now, defining *wellSized* modularly becomes possible:

```
newtype WellSized = WellSized {wellSized :: Bool}
instance (Circuit c, c ≺ Width) ⇒ Circuit (WellSized, c) where
  id n        = (WellSized True, id n)
  fan n       = (WellSized True, fan n)
  above c₁ c₂ = (WellSized (gwellSized c₁ ∧ gwellSized c₂ ∧ gwidth c₁ ≡ gwidth c₂)
                , above (prj c₁) (prj c₂))
  beside c₁ c₂ = (WellSized (gwellSized c₁ ∧ gwellSized c₂), beside (prj c₁) (prj c₂))
  stretch ns c = (WellSized (gwellSized c ∧ length ns ≡ gwidth c), stretch ns (prj c))
gwidth :: (c ≺ Width) ⇒ c → Int
gwidth = width ∘ prj
gwellSized :: (c ≺ WellSized) ⇒ c → Bool
gwellSized = wellSized ∘ prj
```

Essentially, dependent interpretations are still defined using tuples. The dependency on *width* is expressed by constraining the type parameter as $c \prec Width$. Such constraint allows us to simulate the type-refinement of fields in the Scala solution. Although the implementation is modular, it requires some boilerplate. The reuse of *width* interpretation is achieved via delegation, where *prj* needs to be called on each subcircuit. Such explicit delegation simulates the inheritance employed in the Scala solution. Also, auxiliary definitions *gwidth* and *gwellSized* are necessary for projecting the desired interpretations from the constrained type parameter.

3.4.3 Modular terms

As new interpretations may be added later, a problem is how to construct the term that can be interpreted by those new interpretations without reconstruction of the AST for each interpretation. We show how to do this for the circuit shown in Figure 3.2:

```
circuit :: Circuit c ⇒ c
circuit = (fan 2 ‘beside‘ fan 2) ‘above‘
          stretch [2, 2] (fan 2) ‘above‘
          (id 1 ‘beside‘ fan 2 ‘beside‘ id 1)
```

Here, *circuit* is a generic circuit that is not tied to any interpretation. When interpreting *circuit*, its type must be instantiated:

```
> width (circuit :: Width)
4
> depth (circuit :: Depth)
3
> gwellSized (circuit :: (WellSized, Width))
True
```

At user-site, *circuit* must be annotated with the target semantic domain so that an appropriate type class instance for interpretation can be chosen.

Syntax extensions This solution also allows us to modularly extend Scans with more language constructs such as *rstretch*:

```
class Circuit c ⇒ ExtendedCircuit c where
  rstretch :: [Int] → c → c
```

Existing interpretations can be modularly extended to handle *rstretch*:

```
instance ExtendedCircuit Width where
  rstretch = stretch
```

Existing circuits can also be reused for constructing circuits in extended Scans:

```
circuit₂ :: ExtendedCircuit c ⇒ c
circuit₂ = rstretch [2, 2, 2, 2] circuit
```

Table 3.1: *Language features needed for modular interpretations: Scala vs. Haskell.*

Goal	**Scala**	**Haskell**
Multiple interpretation	Trait & Type-refinement	Type class
Interpretation reuse	Inheritance	Delegation
Dependency declaration	Subtyping / Inheritance	Tuples & Type constraints

3.4.4 Comparing Modular Implementations Using Scala and Haskell

Although both the Scala and Haskell solutions are able to model modular dependent interpretations, they use a different set of language features. Table 3.1 compares the language features needed by Scala and Haskell. The Scala approach relies on built-in features. In particular, subtyping, inheritance (mixin composition) and type-refinement are all built-in. This makes it quite natural to program the solutions in Scala, without even needing any form of parametric polymorphism. In contrast, the Haskell solution does not have such built-in support for OO features. Subtyping and type-refinement need to be encoded/simulated using parametric polymorphism and type classes. Inheritance is simulated by explicit delegations. The Haskell encoding is arguably conceptually more difficult to understand and use, but it is still quite simple. One interesting feature that is supported in Haskell is the ability to encode modular terms. This relies on the fact that the constructors are overloaded. The Scala solution presented so far does not allow such overloading, so code using constructors is tied with a specific interpretation. In the next section we will see a final refinement of the Scala solution that enables modular terms, also by using overloaded constructors.

3.5 Modular Terms in Scala

One advantage of the Finally Tagless approach over our Scala approach presented so far is that terms can be constructed modularly without tying those terms to any interpretation. Modular terms are also possible by combining our Scala approach with Object Algebras [Oliveira and Cook, 2012], which employ a technique similar to Finally Tagless in the context of OOP. Differently from the Haskell solution presented in Section 3.4, the Scala approach only employs parametric polymorphism to overload the constructors. Both inheritance and type-refinement do not need to be simulated or encoded.

Object Algebra interface To capture the generic interface of the constructors we define an abstract factory (or Object Algebra interface) for circuits similar to the type class version shown in Section 3.4.1:

```
trait Circuit[C] {
  def id(x : Int) : C
  def fan(x : Int) : C
  def above(x : C, y : C) : C
  def beside(x : C, y : C) : C
  def stretch(x : C, xs : Int*) : C
}
```

which exposes factory methods for each circuit construct supported by Scans.

Abstract terms Modular terms can be constructed via the abstract factory. For example, the circuit shown in Figure 3.2 is built as:

```
def circuit[C](f : Circuit[C]) =
  f.above(f.beside(f.fan(2),f.fan(2)),
          f.above(f.stretch(f.fan(2), 2, 2),
                  f.beside(f.beside(f.id(1),f.fan(2)),f.id(1))))
```

Similarly, *circuit* is a generic method that takes a *Circuit* instance and builds a circuit through that instance. With Scala the definition of *circuit* can be even simpler: we can avoid prefixing "*f*." everywhere by importing *f*. Nevertheless, the definition shown here is more language-independent.

Object Algebras We need concrete factories (Object Algebras) that implement *Circuit* to actually invoke *circuit*. Here is a concrete factory that produces $Circuit_1$:

```
trait Factory1 extends Circuit[Circuit1] {
  def id(x : Int)                           = new Id1      {val n = x}
  def fan(x : Int)                          = new Fan1     {val n = x}
  def above(x : Circuit1, y : Circuit1)     = new Above1   {val c1 = x; val c2 = y}
  def beside(x : Circuit1, y : Circuit1)    = new Beside1  {val c1 = x; val c2 = y}
  def stretch(x : Circuit1, xs : Int*)      = new Stretch1 {val ns = xs.toList; val c = x}
}
```

where the body is identical to the smart constructors presented in Section 3.2.2. Concrete factories for other circuit implementations can be defined in a similar way by instantiating the type parameter *Circuit* accordingly:

```
trait Factory4 extends Circuit[Circuit4] {...}
```

Concrete terms By supplying concrete factories to abstract terms, we obtain concrete terms that can be interpreted differently:

```
circuit(new Factory1 {}).width            // 4
circuit(new Factory4 {}).layout {x ⇒ x}// List(List((0,1),(2,3)),List((1,3)),List((1,2)))
```

Modular extensions Both factories and terms can be *modularly* reused when the DSL is extended with new language constructs. To support right stretch for Scans, we first extend the abstract factory with new factory methods:

```
trait ExtendedCircuit[C] extends Circuit[C] {
  def rstretch(x : C, xs : Int*) : C
}
```

We can also build extended concrete factories upon existing concrete factories:

```
trait ExtendedFactory4 extends ExtendedCircuit[Circuit4] with Factory4 {
  def rstretch(x : Circuit4, xs : Int*) = new RStretch {val c = x; val ns = xs.toList}
}
```

Furthermore, previously defined terms can be reused in constructing extended terms:

```
def circuit2[C](f : ExtendedCircuit[C]) = f.rstretch(circuit(f), 2, 2, 2, 2)
```

3.6 Case Study: A Shallow EDSL for SQL Queries

A common motivation for using deep embeddings is performance. Deep embeddings enable complex AST transformations, which is useful to implement optimizations that improve the performance. An alternative way to obtain performance is to use staging frameworks, such as Lightweight Modular Staging (LMS) [Rompf and Odersky, 2010]. As illustrated by Rompf and Amin [2015] staging can preclude the need for AST transformations for a realistic query DSL. To further illustrate the applicability of shallow OO embeddings, we refactored Rompf and Amin's [2015]'s deep, external DSL implementation to make it more *modular*, *shallow* and *embedded*. The shallow DSL retains the performance of the original deep DSL by generating the same code.

3.6.1 Overview

SQL is the best-known DSL for data queries. Rompf and Amin [2015] present a SQL query processor implementation in Scala. Their implementation is an *external* DSL, which first parses a SQL query into a relational algebra AST and then executes the query in terms of that AST. Based on the LMS framework [Rompf and Odersky, 2010], the SQL compilers are nearly as simple as an interpreter while having performance comparable to hand-written code. The implementation uses deep embedding techniques such as algebraic data types (*case classes* in Scala) and pattern matching for representing and interpreting ASTs. These techniques are a natural choice as multiple interpretations are needed for supporting different backends. But problems arise when the implementation evolves with new language constructs. All existing interpretations have to be modified for dealing with these new cases, suffering from the Expression Problem.

We refactored Rompf and Amin [2015]'s implementation into a shallow EDSL for the following reasons. Firstly, multiple interpretations are no longer a problem for our shallow embedding technique. Secondly, the original implementation contains no hand-coded

AST transformations. Thirdly, it is common to embed SQL into a general purpose language.

To illustrate our shallow EDSL, suppose there is a data file *talks.csv* that contains a list of talks with time, title and room. We can write several sample queries on this file with our EDSL. A simple query that lists all items in *talks.csv* is:

```
def q0 = FROM("talks.csv")
```

Another query that finds all talks at 9 am with their room and title selected is:

```
def q1 = q0 WHERE 'time === "09:00 AM" SELECT('room, 'title)
```

Yet another relatively complex query to find all conflicting talks that happen at the same time in the same room with different titles is:

```
def q2 = q0 SELECT('time, 'room, 'title AS 'title1)   JOIN
            (q0 SELECT('time, 'room, 'title AS 'title2)) WHERE
             'title1 <> 'title2
```

Compared to an external implementation, our embedded implementation has the benefit of reusing the mechanisms provided by the host language for free. As illustrated by the sample queries above, we are able to reuse common subqueries (q_0) in building complex queries (q_1 and q_2). This improves the readability and modularity of the embedded programs.

3.6.2 Embedded Syntax

Thanks to the good support for EDSLs in Scala, we can precisely model the syntax of SQL. The syntax of our EDSL is close to that of LINQ [Meijer et al., 2006], where **select** is an optional, terminating clause of a query. We employ well-established OO and Scala techniques to simulate the syntax of SQL queries in our shallow EDSL implementation. Specifically, we use the *Pimp My Library* pattern [Odersky, 2006] for lifting field names and literals implicitly. For the syntax of combinators such as **where** and **join**, we adopt a fluent interface style. Fluent interfaces enable writing something like "*FROM* (...).*WHERE* (...).*SELECT* (...)". Scala's infix notation further omits "." in method chains. Other famous embedded SQL implementations in OOP such as LINQ [Meijer et al., 2006] also adopt similar techniques in designing their syntax. The syntax is implemented in a pluggable way, in the sense that the semantics is decoupled from the syntax. Details of the syntax implementation are beyond the scope of this thesis. The interested reader can consult the companion code.

Beneath the surface syntax, a relational algebra operator structure is constructed. For example, we will get the following operator structure for q_1:

```
Project(Schema("room", "title"),
           Filter(Eq(Field("time"), Value("09:00 AM")),
                  Scan("talks.csv")))
```

3.6.3 A Relational Algebra Compiler

A SQL query can be represented by a relational algebra expression. The basic interface of operators is modeled as follows:

```
trait Operator {
  def resultSchema : Schema
  def execOp (yld : Record ⇒ Unit) : Unit
}
```

Two interpretations, *resultSchema* and *execOp*, need to be implemented for each concrete operator: the former collects a schema for projection; the latter executes actions to the records of the table. Very much like the interpretation *layout* discussed in Section 3.3.3, *execOp* is both *context-sensitive* and *dependent*: it takes a callback *yld* and accumulates what the operator does to records into *yld* and uses *resultSchema* in displaying execution results. In our implementation *execOp* is indeed introduced as an extension just like *layout*. Here we merge the two interpretations for conciseness of presentation. Some core concrete relational algebra operators are given below:

```
trait Project extends Operator {
  val out, in : Schema; val op : Operator
  def resultSchema = out
  def execOp (yld : Record ⇒ Unit) = op.execOp {rec ⇒ yld (Record (rec (in ), out))}
}
trait Join extends Operator {
  val op1, op2 : Operator
  def resultSchema = op1.resultSchema ++ op2.resultSchema
  def execOp (yld : Record ⇒ Unit) =
    op1.execOp {rec1 ⇒
      op2.execOp {rec2 ⇒
        val keys = rec1.schema intersect rec2.schema
        if (rec1 (keys) ≡ rec2 (keys))
          yld (Record (rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema))
      }
    }
}
trait Filter extends Operator {
  val pred : Predicate; val op : Operator
  def resultSchema = op.resultSchema
  def execOp (yld : Record ⇒ Unit) = op.execOp {rec ⇒ if (pred.eval (rec)) yld (rec)}
}
```

Project rearranges the fields of a record; *Join* matches a record against another and combines the two records if their common fields share the same values; *Filter* keeps a record only when it meets a certain predicate. There are also two utility operators, *Print* and *Scan*, for processing inputs and outputs, whose definitions are omitted for space reasons.

From an interpreter to a compiler The query processor presented so far is elegant but unfortunately slow. To achieve better performance, Rompf and Amin extend the SQL processor in various ways. One direction is to turn the slow query interpreter into a fast query compiler by generating specialized low-level code for a given query. With the help of the LMS framework, this task becomes rather easy. LMS provides a type constructor *Rep* for annotating computations that are to be performed in the next stage. The signature of the staged *execOp* is:

```
def execOp(yld : Record ⇒ Rep[Unit]) : Rep[Unit]
```

where *Unit* is lifted as *Rep*[*Unit*] for delaying the actions on records to the generated code. Two staged versions of *execOp* are introduced for generating Scala and C code respectively. By using the technique presented in Section 3.3, they are added modularly with existing interpretations such as *resultSchema* reused. The implementation of staged *execOp* is similar to the unstaged counterpart except for minor API differences between staged and unstaged types. Hence the simplicity of the implementation remains. At the same time, dramatic speedups are obtained by switching from interpretation to compilation.

Language extensions Rompf and Amin also extend the query processor with two new language constructs, hash joins and aggregates. The introduction of these constructs is done in a modular manner with our approach:

```
trait Group extends Operator {
  val keys, agg : Schema; val op : Operator
  def resultSchema = keys ++ agg
  def execOp(yld : Record ⇒ Unit) {...}
}
trait HashJoin extends Join {
  override def execOp(yld : Record ⇒ Unit) = {
    val keys = op1.resultSchema intersect op2.resultSchema
    val hm = new HashMapBuffer(keys, op1.resultSchema)
    op1.execOp {rec1 ⇒
      hm(rec1(keys)) += rec1.fields
    }
    op2.execOp {rec2 ⇒
      hm(rec2(keys)) foreach {rec1 ⇒
        yld(Record(rec1.fields ++ rec2.fields, rec1.schema ++ rec2.schema))
      }
    }
  }
}
```

Table 3.2: *SLOC for original (Deep) and refactored (Shallow) versions.*

Source	**Functionality**	**Deep**	**Shallow**
query_unstaged	SQL interpreter	83	98
query_staged	SQL to Scala compiler	179	194
query_optc	SQL to C compiler	245	262

Group supports SQL's **group by** clause, which partitions records and sums up specified fields from the composed operator. *HashJoin* is a replacement for *Join*, which uses a hash-based implementation instead of naive nested loops. With inheritance and method overriding, we are able to reuse the field declarations and other interpretations from *Join*.

3.6.4 Evaluation

We evaluate our refactored shallow implementation with respect to the original deep implementation. Both implementations of the DSL (the original and our refactored version) *generate the same code*, thus the performance of the two implementations is similar. Hence we compare the two implementations only in terms of the source lines of code (SLOC). We exclude the code related to surface syntax for the fairness of comparison because our refactored version uses embedded syntax whereas the original uses a parser. As seen in Table 3.2, our shallow approach takes a dozen more lines of code than the original deep approach for each version of SQL processor. The SLOC expansion is attributed to the fact that functional decomposition (case classes) is more compact than object-oriented decomposition in Scala. Nevertheless, our shallow approach makes it easier to add new language constructs.

3.7 Conclusion

This chapter reveals the close correspondence between OOP and shallow embeddings: the essence of both is procedural abstraction. It also showed how OOP increases the modularity of shallow EDSLs. OOP abstractions, including subtyping, inheritance, and type-refinement, bring extra modularity to traditional procedural abstraction. As a result, multiple interpretations are allowed to co-exist in shallow embeddings. Moreover, the multiple interpretations can be *dependent*: an interpretation can depend not only on itself but also on other modular interpretations. Thus the approach presented here allows us to go *beyond simple compositionality*, where interpretations can only depend on themselves.

It has always been a hard choice between shallow and deep embeddings when designing an EDSL: there are some tradeoffs between the two styles. Deep embeddings trade some simplicity and the ability to add new language constructs for some extra power. This extra power enables multiple interpretations, as well as complex AST transformations. As this chapter shows, in languages with OOP mechanisms, multiple (possibly dependent) interpretations are still easy to do with shallow embeddings and the full benefits of an extended form of compositionality still apply. Therefore the motivation to employ deep embeddings becomes weaker than before and mostly reduced to the need for AST transformations. Prior work on the Finally Tagless [Kiselyov, 2012] and Object Algebras [Zhang et al., 2015] approaches already show that AST transformations are still possible in those styles. However this requires some extra machinery, and the line between shallow and deep embeddings becomes quite blurry at that point.

Finally, this work shows a combination of two previously studied solutions to the Expression Problem in OO: the extensible interpreter pattern proposed by Wang and Oliveira [2016] and Object Algebras [Oliveira and Cook, 2012]. The combination exploits the advantages of each of the approaches to overcome the limitations of each approach individually. In the original approach by Wang and Oliveira [2016] modular terms are hard to model, whereas with Object Algebras a difficulty is modeling modular dependent operations. A closely related technique is employed by Cazzola and Vacchi [2016], although in the context of external DSLs. Their technique is slightly different with respect to the extensible interpreter pattern. Essentially while our approach is purely based on subtyping and type-refinement, they use generic types instead to simulate the type-refinement. While the focus of our work is embedded DSLs, the techniques discussed here are useful for other applications, including external DSLs as Cazzola and Vacchi [2016] show.

Chapter 4

Castor: Programming with Extensible Generative Visitors

In this chapter, we turn to metaprogramming-based design patterns. With metaprogramming, the boilerplate associated with design patterns can be largely eliminated. This allows us to employ more powerful design patterns in solving more challenging modularity issues such as binary and producer operations without worrying too much about the complexity brought by these advanced design patterns. We design the Castor framework for programming with extensible generative visitors. Castor supports both functional and imperative style of visitors, pattern matching, type-safe interpreters and graphs. The applicability of Castor is demonstrated by two case studies.

4.1 Introduction

This chapter presents Castor: an extensible and expressive Scala visitor framework. Unlike previous work, Castor aims to support not only a functional style but also an imperative programming style with visitors. Castor visitors bring several advantages over existing approaches:

Concise notation Programming with the Visitor pattern is typically associated with a lot of boilerplate code. *Extensible Visitors* [Zenger and Odersky, 2005; Oliveira, 2009; Hofer and Ostermann, 2010; Zhang and Oliveira, 2017] make the situation even worse due to the heavy use of sophisticated type system features. Although previous work on EVF [Zhang and Oliveira, 2017] alleviated the burden of programmers by generating boilerplate code related to visitors and traversals, it is restricted by Java's syntax and annotation processor. Castor improves on EVF by employing Scala's concise syntax and Scalameta[1] to simplify client code. Unlike Java's annotation processor, we can directly transform the client code with Scalameta, hiding the boilerplate and sophisticated type system features from users.

Pattern matching support Castor comes with support for (type-safe) pattern matching to complement its visitors with a concise notation to express operations. We identify

[1] http://scalameta.org

several desirable properties for pattern matching in an OOP context and show how existing approaches are lacking some of these properties (Section 4.2). We argue that the traditional semantics of pattern matching, which is based on the *order* of patterns and adopted by many approaches, conflicts with the openness of data structures. Therefore we suggest that a more restricted, top-level pattern matching model, where the order of patterns is irrelevant. To compensate for the absence of ordered patterns we propose a complementary mechanism for case analysis with defaults, which can be used when nested or multiple case analysis is needed.

GADT-Style definitions CASTOR supports type-safe interpreters (à la *Finally Tagless*), but with additional support for pattern matching and a generally recursive style. While *Finally Tagless* interpreters are nowadays widely used by programmers in multiple languages (including Haskell and Scala), they must be written in fold-like style. Supporting operations that require nested patterns, or simply depend on other operations is quite cumbersome (although workarounds exist [Kiselyov, 2012]), especially if modularity is to be preserved. In contrast, CASTOR can support those features naturally.

Hierarchical datatypes Functional datatypes are typically flat where variants have no relationships with each other. Object-oriented style datatypes, on the other hand, can be hierarchical [Millstein et al., 2004] where datatype constructors can be refined by more specific constructors. Hierarchical datatypes facilitate reuse since the subtyping relationship allows the semantics defined for supertypes to be reused in subtypes. CASTOR exploits OOP features and employs subtyping to model hierarchical datatypes.

Imperative traversals CASTOR enables many operations to be defined using an imperative style, which is significantly more performant than a functional style (especially in the JVM platform). Both functional and imperative visitors [Buchlovsky and Thielecke, 2006] written with CASTOR are fully extensible and can later support more variants modularly. Imperative visitors enable imperative style traversals that instead of returning a new Abstract Syntax Tree (AST), modify an existing AST in-place.

Graph structures Finally functional techniques usually only support tree structures well, but graph structures are poorly supported. CASTOR supports type-safe extensible programming on graph structures. Compared to trees, graphs are a more general data structure that have many important applications. In the domain of compilers, abstract semantic graphs can be used for representing shared subexpressions, which facilitate optimizations like common subexpression elimination.

In summary, this chapter makes the following contributions:

- **Extensible pattern matching with modular external visitors:** We evaluate existing approaches to pattern matching in an OOP context (Section 4.2). We show how to incoorporate extensible (or open) pattern matching support on modular external visitors, which allows CASTOR to define non-trivial pattern matching operations.

- **Support for hierarchical datatypes:** Besides flat datatypes that are typically modeled in functional languages, we show how OOP style hierarchical datatypes is supported in CASTOR (Section 4.3).

- **Support for GADTs:** We show how to use CASTOR's support for GADTs in building well-typed interpreters (Section 4.4), which would be quite difficult to model in a *Finally Tagless* style.

- **Imperative style modular external visitors:** We show how to define imperative style modular external visitors in CASTOR (Section 4.5).

- **Support for graph structures:** We show how to do type-safe extensible programming on graph structures, which generalize the typical tree structures in functional programming (Section 4.5).

- **The CASTOR framework:** We present a novel encoding for modular pattern matching based on extensible visitors (Section 4.2.7). The encoding is automated using metaprogramming and the transformation is formalized (Section 4.6).

- **Case studies:** We conduct two case studies to illustrate the effectiveness of CASTOR. The first case study on TAPL interpreters (Section 4.7) demonstrates functional aspects of CASTOR, while the second one on UML activity diagrams (Section 4.8) demonstrates the object-oriented aspects of CASTOR.

Source code for CASTOR and case studies is available at:

`https://github.com/wxzh/Castor`

4.2 Open Pattern Matching

Pattern matching is a pervasive and useful feature in functional languages, e.g. ML [Milner et al., 1997] and Haskell [Jones, 2003], for processing data structures conveniently. Data structures are firstly modeled using algebraic datatypes and then processed through pattern matching. On the other hand, OOP uses class hierarchies instead of algebraic datatypes to model data structures. Still, the same need for processing data structures also exists in OOP. However, there are important differences between data structures modeled with algebraic datatypes and class hierarchies. Algebraic datatypes are typically *closed*, having a fixed set of variants. In contrast, class hierarchies are *open*, allowing the addition of new variants. A closed set of variants facilitates exhaustiveness checking of patterns but sacrifices the ability to add new variants. OO class hierarchies do support the addition of new variants, but without mechanisms similar to pattern matching, some programs are unwieldy and cumbersome to write. In this section, we first characterize four desirable properties of pattern matching in the context of OOP. We then review some of the existing pattern matching approaches in OOP and discuss why they fall in short of the desirable properties. This section ends with an overview of CASTOR and an evaluation summary on the presented approaches.

4.2.1 Desirable Properties of Open Pattern Matching

We identify the following desirable properties for pattern matching in an OOP context:

- **Conciseness.** Patterns should be described concisely with potential support for wildcards, deep patterns, and guards.
- **Exhaustiveness.** Patterns should be exhaustive to avoid runtime matching failure. The exhaustiveness of patterns should be statically verified by the compiler and the missing cases should be reported if patterns are incomplete.
- **Extensibility.** Datatypes should be extensible in the sense that new data variants can be added while existing operations can be reused without modification or recompilation.
- **Composability.** Patterns should be composable so that complex patterns can be built from smaller pieces. When composing overlapped patterns, programmers should be warned about possible redundancies.

Using these properties as criteria, we next evaluate pattern matching approaches in OOP. We show that many widely used approaches lack some of these properties. We argue that a problem is that many approaches try to closely follow the traditional semantics of pattern matching, which assumes a closed set of variants. Under a closed set of variants, it is natural to use the *order* of patterns to prioritize some patterns over the others. However, when the set of variants is not predefined a priori then relying on some ordering of patterns is problematic, especially if separate compilation and modular type-checking are to be preserved. Nonetheless, many OO approaches, which try to support both an extensible set of variants and pattern matching, still try to use the order of patterns to define the semantics. Unfortunately, this makes it hard to support other desirable properties such as exhaustiveness or composability.

4.2.2 Running Example: Arith

To facilitate our discussion, a running example from TAPL [Pierce, 2002]—an untyped, arithmetic language called Arith—is used throughout this chapter. The syntax and semantics of Arith are formalized in Figure 4.1. Our goal is to model the syntax and semantics of Arith in a concise and modular manner.

Arith has the following syntactic forms: zero, successor, predecessor, true, false, conditional and zero test. The definition *nv* identifies 0 and successive application of `succ` to 0 as numeric values. The operational semantics of Arith is given in *small-step* style, with a set of reduction rules specifying how a term can be rewritten in one step. Repeatedly applying these rules will eventually evaluate a term to a value. There might be multiple rules defined on a single syntactic form. For instance, rules PredZero, PredSucc and Pred are all defined on a predecessor term. How `pred` t is going to be evaluated in the next step is determined by the shape of the inner term t: if t is 0, then PredZero will be

$$\begin{array}{lcl} t & ::= & \mathtt{0} \mid \mathtt{succ}\ t \mid \mathtt{pred}\ t \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ t \mid \mathtt{iszero}\ t \\ nv & ::= & \mathtt{0} \mid \mathtt{succ}\ nv \end{array}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathtt{succ}\ t_1 \longrightarrow \mathtt{succ}\ t_1'} \qquad \mathtt{pred}\ \mathtt{0} \longrightarrow \mathtt{0}\ \textsc{PredZero} \qquad \mathtt{pred}\ (\mathtt{succ}\ nv_1) \longrightarrow nv_1\ \textsc{PredSucc}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathtt{pred}\ t_1 \longrightarrow \mathtt{pred}\ t_1'}\ \textsc{Pred} \qquad \mathtt{if}\ \mathtt{true}\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \longrightarrow t_2$$

$$\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \longrightarrow t_3 \qquad \frac{t_1 \longrightarrow t_1'}{\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \longrightarrow \mathtt{if}\ t_1'\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3}$$

$$\mathtt{iszero}\ \mathtt{0} \longrightarrow \mathtt{true} \qquad \mathtt{iszero}\ (\mathtt{succ}\ nv_1) \longrightarrow \mathtt{false} \qquad \frac{t_1 \longrightarrow t_1'}{\mathtt{iszero}\ t_1 \longrightarrow \mathtt{iszero}\ t_1'}$$

Figure 4.1: *The syntax and semantics of* ARITH.

applied; if t is a successor application to a numeric value, then PREDSUCC will be applied; otherwise PRED will be applied.

ARITH is a good example for assessing the four properties because: 1) The small-step style semantics is best expressed with a concise *nested case analysis* on terms; 2) ARITH is, in fact, a unification of two sublanguages, NAT (zero, successor and predecessor) and BOOL (true, false, and conditional) plus an extension (zero test). Ideally, NAT and BOOL should be *separately defined* and *modularly reused*.

4.2.3 The VISITOR Pattern

The VISITOR design pattern [Gamma et al., 1994] is frequently used to implement interpreters or compilers because of its ability to add new interpretations or compiler phases without modifying the class hierarchy. Let us implement the ARITH language using the VISITOR pattern step by step. The implementation is written in Scala without using any Scala-specific features and can be easily mapped to other OOP languages like C++ or Java.

Abstract syntax The abstract syntax of ARITH is modeled by the following class hierarchy:

```
abstract class Tm {
  def accept[A](v: TmVisit[A]): A
}
class TmZero() extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmZero(this)
}
class TmSucc(val t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmSucc(this)
}
class TmPred(val t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]) = v.tmPred(this)
}
class TmTrue() extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmTrue(this)
}
class TmFalse extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmFalse(this)
}
class TmIf(val t1: Tm, val t2: Tm, val t3: Tm) extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmIf(this)
}
class TmIsZero(val t: Tm) extends Tm {
  def accept[A](v: TmVisit[A]): A = v.tmIsZero(this)
}
```

The abstract class `Tm` represents the datatype of terms, and syntactic constructs of terms are subclasses of `Tm`. A generic `accept` method is defined throughout the class hierarchy, which is implemented by invoking the corresponding lowercase *visit method* exposed by `TmVisit`.

Visitor interface `TmVisit` is the *visitor interface* that declares all the visit methods required by `accept` implementations. Its definition is given below:

```
trait TmVisit[A] {
  def tmZero(x: TmZero): A
  def tmSucc(x: TmSucc): A
  def tmPred(x: TmPred): A
  def tmTrue(x: TmTrue): A
  def tmFalse(x: TmFalse): A
  def tmIf(x: TmIf): A
  def tmIsZero(x: TmIsZero): A
}
```

`TmVisit` is parameterized by `A` for abstracting over the return type of visit methods. Each visit method takes an instance of its corresponding class and returns a value of `A`.

Concrete visitors Operations over `Tm` are *concrete visitors* that implement the visitor interface `TmVisit`. The numeric value checker is defined like this:

```
class Nv extends TmVisit[Boolean] {
  def tmZero(x: TmZero) = true
  def tmSucc(x: TmSucc)= x.t.accept(this)
  def tmPred(x: TmPred) = false
  def tmTrue(x: TmTrue) = false
  def tmFalse(x: TmFalse) = false
  def tmIf(x: TmIf) = false
  def tmIsZero(x: TmIsZero) = false
}
```

`Nv` implements `TmVisit` by instantiating the type parameter `A` as `Boolean` and giving an implementation to each visit method. Here, the interesting cases are `tmZero` and `tmSucc`. For the former, a `true` is returned; for the latter, we call `_.t.accept(this)` for recursively applying `Nv` to check the inner term. The remaining cases are not numeric values thus return `false`.

With Nv defined, we can now implement the small-step evaluation visitor:

```
class Eval1 extends TmVisit[Tm] {
//  val eval1 = this  // Dependency on the visitor itself
  val nv = new Nv   // Dependency on another visitor
  def tmZero(x: TmZero) = throw NoRuleApplies
  def tmSucc(x: TmSucc) = new TmSucc(x.t.accept(this))
  def tmPred(x: TmPred) = x.t.accept(new TmVisit[Tm] {
    def tmZero(y: TmZero) = y                               // PredZero
    def tmSucc (y: TmSucc) =
      if (y.t.accept(nv)) y.t                               // PredSucc
      else new TmPred(y.t.accept(eval1))                    // Pred
    def tmPred(y: TmPred) = new TmPred(y.accept(eval1))     // Pred
    def tmTrue(y: TmTrue) = new TmPred(y.accept(eval1))     // Pred
    def tmFalse(y: TmFalse) = new TmPred(y.accept(eval1))   // Pred
    def tmIf(y: TmIf) = new TmPred(y.accept(eval1))         // Pred
    def tmIsZero(y: TmIsZero) = new TmPred(y.accept(eval1)) // Pred
  })
  def tmTrue(x: TmTrue) = throw NoRuleApplies
  def tmFalse(x: TmFalse)  = throw NoRuleApplies
  def tmIf(x: TmIf) = x.t1.accept(new TmVisit[Tm] {
    def tmTrue(y: TmTrue) = x.t2
    def tmFalse(y: TmFalse) = x.t3
    def tmZero(y: TmZero) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmSucc(y: TmSucc) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmPred(y: TmPred) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmIf(y: TmIf) = new TmIf(y.accept(eval1),x.t2,x.t3)
    def tmIsZero(y: TmIsZero) = new TmIf(y.accept(eval1),x.t2,x.t3)
  })
  def tmIsZero(x: TmIsZero) = x.t.accept(new TmVisit[Tm] {
    def tmZero(y: TmZero) = new TmTrue
    def tmSucc (y: TmSucc) =
      if (y.t.accept(nv)) new TmFalse
      else new TmIsZero(y.accept(eval1))
    def tmPred(y: TmPred) = new TmIsZero(y.accept(eval1))
    def tmTrue(y: TmTrue) = new TmIsZero(y.accept(eval1))
    def tmFalse(y: TmFalse) = new TmIsZero(y.accept(eval1))
    def tmIf(y: TmIf) = new TmIsZero(y.accept(eval1))
    def tmIsZero(y: TmIsZero) = new TmIsZero(y.accept(eval1))
  })
}
```

The small-step evaluator rewrites a term to another thus A is instantiated as Tm. Since primitive cases are already values, we simply throw a NoRuleApplies exception for tmZero, tmTrue and tmFalse. Defining the case for tmSucc is easy too: we construct a new successor with its inner term rewritten by eval1. In contrast, defining tmPred, tmIf and tmIsZero is trickier because they all have multiple rules. Take tmPred for example. As a visitor recognizes only one level representation of a term, it is insufficient to encode rules that require nested case analysis. To further reveal the shape of the inner term, anonymous visitors are created. Rules like PredSucc can then be specified inside the tmSucc method of the inner visitor. Moreover, the inner visitor of tmPred depends on both Eval1 and Nv. These dependencies are expressed by the fields eval1 and nv, which are instantiated as visitor instances. Then we can pass eval1 or nv as an argument to the accept method for using the dependency. Notice that the Pred rule is repeated 6 times. Similar situations also happen in `tmIf` and `tmIsZero`, making the overall implementation of `Eval1` quite lengthy.

Client code We can write some tests for our implementation of Arith:

```scala
// iszero (if false then true else pred (succ 0))
val tm = new TmIsZero(new TmIf(new TmFalse,new TmTrue,new TmPred(new TmSucc(new
    TmZero))))
val eval1 = new Eval1
val tm1 = tm.accept(eval1)  // iszero (pred (succ 0))
val tm2 = tm1.accept(eval1) // iszero 0
val tm3 = tm2.accept(eval1) // 0
```

where we construct a term using all syntactic forms of the Arith language and evaluate it step by step using `eval1`. The evaluation result of each step is shown in the comments on the right hand side.

Discussion of the approach The conventional Visitor pattern has been criticized for its *verbosity* and *inextensibility* [Meyer and Arnout, 2006; Pati and Hill, 2014], which are manifested in the implementation of Arith. Programming with the Visitor pattern is associated with a lot of infrastructure code, including the visitor interface, the class hierarchy, etc. Writing such infrastructure manually is tedious and error-prone, especially when there are many classes involved. Such verbosity restricts the usage Visitor pattern, as Martin [2002] wrote:

> "*Often, something that can be solved with a* Visitor *can also be solved by something simpler.*"

Moreover, the Visitor pattern suffers from the Expression Problem [Wadler, 1998]: it is *easy* to add new operations by defining new visitors (as illustrated by `nv` and `eval1`) but *hard* to add new variants. The reason is that `Tm` and `TmVisit` are tightly coupled. When trying to add new subclasses to the `Tm` hierarchy, it is not possible to implement their `accept` methods because there exist no corresponding visit methods in `TmVisit`. A non-solution is to modify `TmVisit` with new visit methods. As a consequence, all existing concrete implementations of `TmVisit` have to be modified in order to account for those variants. This violates the "*no modification on existing code*" principle of the Expression Problem. Modification is even impossible if the source code is unavailable. As a result, Nat and Bool cannot be separated from Arith. Thus, the whole implementation is neither extensible nor composable. Nevertheless, the exhaustiveness on visit methods is guaranteed since a class cannot contain any abstract methods.

4.2.4 Sealed Case Classes

The Visitor pattern is often used as a poor man's approach to pattern matching in OO languages. Fortunately, Scala [Odersky et al., 2004] is a language that unifies functional and OO paradigms and supports pattern matching natively via case classes/extractors [Emir et al., 2007]. Case classes can be either open or sealed. Sealed case classes are close to algebraic datatypes in functional languages, which have a fixed set of variants.

Representing the Tm hierarchy using sealed case classes looks like this:

```
sealed trait Tm
case object TmZero extends Tm
case class TmSucc(t: Tm) extends Tm
case class TmPred(t: Tm) extends Tm
case object TmTrue extends Tm
case object TmFalse extends Tm
case class TmIf(t1: Tm, t2: Tm, t3: Tm) extends Tm
case class TmIsZero(t: Tm) extends Tm
```

The differences are that Tm is a sealed trait and variants of Tm are additionally marked as case. Also, no-argument variants are Scala's singleton objects and fields of case classes are by default val.

The case keyword triggers the Scala compiler to automatically inject methods into a class, including a constructor method (apply) and an extractor method (unapply). The injected constructor method simplifies creating objects from case classes. For example, a successor application to zero can be constructed via TmSucc(TmZero). Conversely, the injected extractor enables tearing down an object via pattern matching.

The numeric value checker can be defined by pattern matching on the term:

```
def nv(t: Tm): Boolean = t match {
  case TmZero => true
  case TmSucc(t1) => nv(t1)
  case _ => false
}
```

The term t is matched sequentially against a series of patterns (case clauses). For example, TmSucc(TmZero) will be handled by the second case clause of nv, which recursively invokes nv on its subterm t1 (which is TmZero). Then, TmTrue will be matched by the first case clause with a true returned eventually. A *wildcard pattern* (_) is used in the last case clause for handling boring cases altogether.

The strength of pattern matching shines in encoding the small-step semantics:

```
def eval1(t: Tm): Tm = t match {
  case TmSucc(t1) => TmSucc(eval1(t1))
  case TmPred(TmZero) => TmZero            // PredZero
  case TmPred(TmSucc(t1)) if nv(t1) => t1 // PredSucc
  case TmPred(t1) => TmPred(eval1(t1))     // Pred
  case TmIf(TmTrue,t2,_) => t2
  case TmIf(TmFalse,_,t3) => t3
  case TmIf(t1,t2,t3) => TmIf(eval1(t1),t2,t3)
  case TmIsZero(TmZero) => TmTrue
  case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
  case TmIsZero(t1) => TmIsZero(eval1(t1))
  case _ => throw NoRuleApplies
}
```

With the help of pattern matching, the overall definition is a direct mapping from the formalization shown in Figure 4.1. There is a one-to-one correspondence between the rules and the case clauses. For example, PredSucc is concisely described by a *deep* pattern (`TmPred(TmSucc(t1))`) with a *guard* (`if nv(t1)`) and Pred is captured only once by `TmPred(t1)`.

Client code The client code is also more natural and compact than that in visitors:

```
val tm = TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))
val tm1 = eval1(tm)  // iszero (pred (succ 0))
val tm2 = eval1(tm1) // iszero 0
val tm3 = eval1(tm2) // 0
```

where `new` clauses are no longer needed.

Discussion of the approach The Arith implementation using sealed case classes is very concise. Moreover, sealed case classes facilitate exhaustiveness checking on patterns since all variants are statically known. If we forgot to write the wildcard pattern in `nv`, the Scala compiler would warn us that a `case` clause for `TmPred` is missing. An exception is `eval1`, whose exhaustiveness is not checked by the compiler due to the use of guards. The reason is that a guard might call some function whose execution result is only known at runtime, making the reachability of that pattern difficult to decide statically. The price to pay for exhaustiveness is the inability to add new variants of `Tm` in separate files. Thus, like the visitor version, the implementation is neither extensible nor composable.

4.2.5 Open Case Classes

While the implementation using sealed case classes is concise, it is not modular because Arith is still defined as a whole. To separate out Nat and Bool, we turn to open case classes by trading exhaustiveness checking for the ability to add new variants in separate files. To make up for the loss of exhaustiveness, Zenger and Odersky's [2001] idea of *Extensible Algebraic Datatypes with Defaults* (EADDs) can be applied. The key idea is to always use a default in each operation to handle variants that are not explicitly mentioned. The existence of a default makes operations extensible, as variants added later will be automatically subsumed by that default. If the extended variants have behavior different from the default, we can define a new operation that deals with the extended variants and delegates to the old operation.

We first remove the `sealed` constraint on `Tm` and specify the default behavior of `eval1` inside a trait `Term`:

```
trait Term {
  trait Tm
  def eval1(t: Tm): Tm = throw NoRuleApplies
}
```

Then, Nat can be defined as an extended trait for `Term`:

```
trait Nat extends Term {
  case object TmZero extends Tm
  case class TmSucc(t: Tm) extends Tm
  case class TmPred(t: Tm) extends Tm
  def nv(t: Tm): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  override def eval1(t: Tm): Tm = t match {
    case TmSucc(t1) => TmSucc(eval1(t1))
    case TmPred(TmZero) => TmZero           // PredZero
    case TmPred(TmSucc(t1)) if nv(t1) => t1 // PredSucc
    case TmPred(t1) => TmPred(eval1(t1))    // Pred
    case _ => super.eval1(t)
  }
}
```

`Nat` introduces `TmZero`, `TmSucc` and `TmPred` as variants of `Tm`. `nv` is defined in the old way. `eval1` is overridden with case clauses for `TmSucc` and `TmPred`, and `TmZero` is dealt by `Term`'s `eval1` via a `super` call.

Similarly, Bool is defined as another trait that extends `Tm` with its own variants and `eval1`:

```
trait Bool extends Term {
  case object TmTrue extends Tm
  case object TmFalse extends Tm
  case class TmIf(t1: Tm,t2: Tm,t3: Tm) extends Tm
  override def eval1(t: Tm): Tm = t match {
    case TmIf(TmTrue,t2,_) => t2
    case TmIf(TmFalse,_,t3) => t3
    case TmIf(t1,t2,t3) => TmIf(eval1(t1),t2,t3)
    case _ => super.eval1(t)
  }
}
```

Finally, Arith can be defined as a unification of Nat and Bool implementations:

```
trait Arith extends Nat with Bool {
  case class TmIsZero(t: Tm) extends Tm
  override def eval1(t: Tm) = t match {
    case TmIsZero(TmZero) => TmTrue
    case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
    case TmIsZero(t1) => TmIsZero(eval1(t1))
    case TmZero => super[Nat].eval1(t)
    case _: TmSucc => super[Nat].eval1(t)
    case _: TmPred => super[Nat].eval1(t)
    case _ => super[Bool].eval1(t)
  }
}
```

Scala's *mixin composition* allows `Arith` to extend both `Nat` and `Bool`. The definition `nv` inherited from `Nat` works well in `Arith`, as it happens to have a very good default that automatically fits for the new cases. For instance, calling `nv(TmFalse)` returns `false` as expected. However, overridding `eval1` becomes problematic. We cannot simply complement the cases for `TmIsZero` and handle all the inherited cases at once since both `Nat` and `Bool` are extended. Instead we have to separate the inherited cases using *typecases* and delegate appropriately to either `Nat` or `Bool` via `super` calls.

Discussion of the approach Combining open case classes with EADDs brings extensibility. This idea works well for *linear* extensions (such as Nat and Bool) but not so well for *non-linear* extensions like Arith. As shown by `eval1` in `Arith`, composing non-linear extensions is tedious and error-prone. Without any assistance from the Scala compiler during this process, it is rather easy to make mistakes like forgetting to delegate a case or delegating a case to a wrong parent. Moreover, the exhaustiveness checking on case clauses is lost. Although in the spirit of EADDs case clauses should always end with a wildcard that ensures exhaustiveness, it is not enforced by the Scala compiler.

4.2.6 Partial Functions

To ease the composition of `Nat` and `Bool`, one may consider Scala's `PartialFunction`. `PartialFunction` provides an `orElse` method for composing partial functions. `orElse` tries the composed partial functions sequentially until no `MatchError` is raised.

The open case class version of Arith can be adapted to a partial function version with a few changes. First, `eval1` in `Term` should be declared as a partial function:

```
def eval1: PartialFunction[Tm,Tm]
```

Second, wildcards cannot be used in implementing `eval1` anymore because they will shadow other partial functions to be composed. For example, `eval1` in `Bool` is rewritten as:

```
override def eval1 {
  case TmIf(TmTrue,t2,_) => t2
  case TmIf(TmFalse,_,t3) => t3
  case TmIf(t1,t2,t3) => TmIf(eval1(t1),t2,t3)
  case TmTrue => throw NoRuleApplies
  case TmFalse => throw NoRuleApplies
}
```

An instance of `PartialFunction[Tm,Tm]` is constructed using the anonymous function syntax with the argument `Tm` being directly pattern matched. The wildcard pattern is replaced by two constructor patterns `TmTrue` and `TmFalse` with identical right hand side, losing some convenience. Nevertheless, partial functions make the composition work more smoothly, avoiding the problems caused by the open case classes approach:

```
override def eval1 = super[Nat].eval1 orElse super[Bool].eval1 orElse {
  case TmIsZero(TmZero) => TmTrue
  case TmIsZero(TmSucc(t1)) if nv(t1) => TmFalse
  case TmIsZero(t1) => TmIsZero(eval1(t1))
}
```

`eval1` is overridden by chaining `eval1` from `Nat` and `Bool` as well as a new partial function for the zero test using the `orElse` combinator.

Discussion of the approach Although combining open case classes with partial functions makes the composition smoother, it is still not fully satisfactory. The `orElse` combinator is left-biased, thus the *composition order determines the composed semantics*. That is, `f orElse g` is not equivalent to `g orElse f`, if `f` and `g` are two overlapped partial functions (i.e. containing `case` clauses with identical left hand side but different right hand side). When composing such overlapped partial functions, `orElse` gives no warning. Also, the semantics of the overlapped patterns are all from either `f` or `g`, depending on which comes first. It is not possible to have a mixed semantics for overlapped patterns (e.g. picking `case A` from `f` and `case B` from `g` when both `f` and `g` define `case A` and `case B`), which restricts the reusability of partial functions. Lastly, partial functions rely on exception handling, which has a negative impact on performance.

4.2.7 Extensible Visitors

Essentially what makes pattern matching hard to be extended or composed is the *order-sensitive* semantics of pattern matching and wildcard patterns that cover both known and unknown variants. We think it is useful to distinguish between top-level (shallow) patterns and nested (deep) patterns. Top-level patterns should be order-insensitive and partitioned into multiple definitions so that they can be easily composed. We can achieve this by combining open case classes with extensible visitors [Zenger and Odersky, 2005; Oliveira, 2009; Hofer and Ostermann, 2010; Zhang and Oliveira, 2017].

The Arith implementation is organized in a way similar to the open case classes approach. Let us start with `Term`:

```
trait Term {
  type TmV <: TmVisit
  trait Tm { def accept(v: TmV): v.OTm }
  trait TmVisit { _: TmV =>
    type OTm
    def apply(t: Tm) = t.accept(this)
  }
  trait TmDefault extends TmVisit { _: TmV =>
    def tm: Tm => OTm
  }
  trait Eval1 extends TmDefault { _: TmV =>
    type OTm = Tm
    def tm = _ => throw NoRuleApplies
  }
  val eval1: Eval1
}
```

Instead of using `TmVisit` in declaring the `accept` method, we use an *abstract type member* `TmV` and constrain it to be a *subtype* of `TmVisit`. This enables invocations on the methods declared inside `TmVisit`, but at the same time, decouples `Tm` from `TmVisit`. The upper bound of the return type of the visit methods is also captured by an abstract type rather than a type parameter for avoiding reinstantiation in inherited visitors. Accordingly, the return type of `accept` is now a path dependent type `v.OTm`. A syntactic sugar method `apply` is defined inside `TmVisit` for enabling `v(x)` as a shorthand of `x.accept(v)`, where `x` and `v` are instances of `Tm` and `TmVisit`, respectively. To pass `this` as an argument of `accept` in implementing `apply`, we state that `TmVisit` is of type `TmV` using a *self-type annotation*. To mimic wildcards, we use default visitors [Nordberg III, 1996]. But unlike wildcards, default visitors only deal with *known* variants. TmDefault is the default visitor interface, which extends TmVisit with a generic tm method for specifying the default behavior. Eval1 is a default visitor thus it extends TmDefault, specifies the output type OTm as Tm and implements tm. Each concrete visitor has a companion **val** declaration for allowing themselves to be used in other visitors.

The encoding makes more sense with the implementation of Nat given:

```
trait Nat extends Term {
  type TmV <: TmVisit
  case object TmZero extends Tm {
    def accept(v: TmV): v.OTm = v.tmZero
  }
  case class TmSucc(t: Tm) extends Tm {
    def accept(v: TmV): v.OTm = v.tmSucc(this)
  }
  case class TmPred(t: Tm) extends Tm {
    def accept(v: TmV): v.OTm = v.tmPred(this)
  }
  trait TmVisit extends super.TmVisit { _: TmV =>
    def tmZero: OTm
    def tmSucc: TmSucc => OTm
    def tmPred: TmPred => OTm
  }
  trait TmDefault extends TmVisit with super.TmDefault { _: TmV =>
    def tmZero = tm(TmZero)
    def tmSucc = tm(_)
    def tmPred = tm(_)
  }
  def nv(t: Tm): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  trait Eval1 extends TmDefault with super.Eval1 { _: TmV =>
    override def tmSucc = x => TmSucc(this(x.t))
    override def tmPred = {
      case TmPred(TmZero) => TmZero
      case TmPred(TmSucc(t)) if nv(t) => t
      case TmPred(t) => TmPred(this(t))
    }
  }
}
```

Tm is extended with several case classes/objects. Correspondingly TmVisit is extended with new visit methods and TmV is *covariantly refined* as the subtype of the extended TmVisit. Visit methods are declared using Scala's functions instead of ordinary methods for two reasons. First, the argument type (e.g. TmSucc) has already been revealed by the method name (tmSucc) and can be inferred by the Scala compiler without losing information. Second, first-class functions facilitate pattern matching on the argument. These two advantages result in a concise definition of Eval1, where the type of x is omitted and a value of TmPred => Tm is constructed by pattern matching. Unlike conventional visitors, nested case analysis is much simplified via (nested) pattern matching rather than auxiliary visitors. For example, when a predecessor term is processed by `Eval1`, it will be recognized and dispatched to the `tmPred` method. Then the `TmPred` object is matched by the `case` clauses. As these are `case` clauses, deep patterns and guards can be used. To restore the convenience of wildcards for top-level patterns, `TmDefault` is used, which implements visit methods by delegating to `tm`. Notice that `Eval1` is defined as a trait instead of a class for enabling mixin composition. By extending both `TmDefault` and `super.Eval1`, `Eval1` only needs to override interesting cases.

The numeric value checker is defined as a method rather than a visitor. This is because, as we have discussed, *nv* is a good candidate for applying EADDs. Of course, `nv` can be defined as a default visitor like `Eval1`. But whenever `Nat` is extended with new terms, the definition of *nv* has to be refined by composing `Nv` with the extended `TmDefault`.

Bool is defined in a similar manner:

```
trait Bool extends Term {
  type TmV <: TmVisit
  trait TmVisit extends super.TmVisit { _: TmV =>
    def tmTrue: OTm
    def tmFalse: OTm
    def tmIf: TmIf => OTm
  }
  trait TmDefault extends TmVisit with super.TmDefault { _: TmV =>
    def tmTrue = tm(TmTrue)
    def tmFalse = tm(TmFalse)
    def tmIf = tm
  }
  case object TmTrue extends Tm {
    override def accept(v: TmV) = v.tmTrue
  }
  case object TmFalse extends Tm {
    override def accept(v: TmV) = v.tmFalse
  }
  case class TmIf(t1: Tm, t2: Tm, t3: Tm) extends Tm {
    override def accept(v: TmV) = v.tmIf(this)
  }
  trait Eval1 extends TmDefault with super.Eval1 { _: TmV =>
    override def tmIf = {
      case TmIf(TmTrue,t2,_) => t2
      case TmIf(TmFalse,_,t3) => t3
      case TmIf(t1,t2,t3) => TmIf(this(t1), t2, t3)
    }
  }
}
```

With case clauses partitioned into visit methods according to their top-level pattern, unifying `Nat` and `Bool` becomes easy via Scala's mixin composition:

```
trait Arith extends Nat with Bool {
  type TmV <: TmVisit
  case class TmIsZero(t: Tm) extends Tm {
    override def accept(v: TmV) = v.tmIsZero(this)
  }
  trait TmVisit extends super[Nat].TmVisit
    with super[Bool].TmVisit { _: TmV =>
    def tmIsZero: TmIsZero => OTm
  }
  trait TmDefault extends TmVisit with super[Nat].TmDefault
    with super[Bool].TmDefault { _: TmV =>
    def tmIsZero = tm
  }
  trait Eval1 extends TmVisit with super[Nat].Eval1
    with super[Bool].Eval1 { _: TmV =>
    def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
  }
}
```

Defining Eval1 for Arith only needs to inherit Eval1 definitions from Nat and Bool and complement the tmIsZero method. Since tmIsZero is an interesting case, Eval1 extends TmVisit rather than TmDefault.

Instantiation Components defined in this way cannot be directly used in client code. An additional step to instantiate traits into objects is required. Instantiating Arith, for example, is done like this:

```
object Arith extends Arith {
  type TmV = TmVisit
  object eval1 extends Eval1
}
```

The companion object Arith binds the abstract type TmV to its corresponding the visitor interface TmVisit. The eval1 declaration is met by a singleton object that extends Eval1. If Eval1 does not implement all the visit methods, the object creation fails, with the missing methods reported.

Client code Now we can use the companion object Arith in client code:

```
import Arith._
val tm = TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))
val tm1 = eval1(tm)  // iszero (pred (succ 0))
val tm2 = eval1(tm1) // iszero 0
val tm3 = eval1(tm2) // 0
```

By importing Arith, the constructors and visitors defined inside Arith are in scope. With the syntactic sugar defined for visitors, a term can be constructed and evaluated identically to the case class version.

Discussion of the approach With the powerful extensible visitor encoding, the Arith implementation is made both extensible and composable. However, extensible visitors are even more verbose than conventional ones. The use of traits in implementing visitors brings composability but, at the same time, requires extra instantiation code. Another downside of using traits is that the exhaustiveness checking on visit methods is deferred to the instantiation stage. Moreover, the encoding relies on advanced features of Scala, making it less accessible to novice Scala programmers.

4.2.8 EVF

Programming with visitors can be greatly simplified with the associated infrastructure automatically generated. This idea has been adopted in our previous work on EVF [Zhang and Oliveira, 2017], which employs Java annotation processors for generating extensible visitor infrastructure.

EVF uses Object Algebra interfaces [Oliveira and Cook, 2012] to describe the abstract syntax:

```
@Visitor interface TmAlg<Tm> {
  Tm TmZero();
  Tm TmSucc(Tm t);
  Tm TmPred(Tm t);
}
```

where the type parameter `Tm` represents the datatype and capitalized methods that return `Tm` represent variants of `Tm`. Annotated as `@Visitor`, `TmAlg` will be recognized and processed by EVF. Then the infrastructure for `TmAlg` will be generated, including a class hierarchy, a visitor interface and various default visitors. Based on the generated visitor infrastructure, we are able to implement `Nv`:

```
interface Nv<Tm> extends TmAlgDefault<Tm,Boolean> {
  @Override default Zero<Boolean> m() {
    return () -> false;
  }
  default Boolean TmZero() {
    return true;
  }
  default Boolean TmSucc(Tm t) {
    return visitTm(t);
  }
}
```

`Nv` is defined as an interface with visit methods implemented using default methods for retaining composability. The Java extensible visitor encoding adopted by EVF is, however, not as powerful as the Scala one shown in Section 4.2.7, which does not support modular ASTs. Whenever an annotated Object Algebra interface gets extended, a new class hierarchy is generated. Thus, we cannot refer to a concrete datatype directly in visitors since this will make them inextensible. Instead, datatypes are kept abstract in visitors. To traverse an abstract datatype like `Tm`, `visitTm` is called. `visitTm` is a method exposed by the generated visitor interface, similar to `apply` shown in Section 4.2.7. `TmAlgDefault` is the default visitor similar to `TmDefault`, where the default behavior is specified inside `m()`.

Defining `Eval1` is tricker:

```
interface Eval1<Tm> extends TmAlgDefault<Tm,Tm>, tm.Eval1<Tm> {
  TmAlgMatcher<Tm,Tm> matcher(); // Dependency for nested case analysis
  TmAlg<Tm> f();                 // Dependency for AST reconstruction
  Nv<Tm> nv();                   // Dependency for another visitor
  @default Tm TmPred(Tm t) {
    return matcher()
        .TmZero(() -> t)
        .TmSucc(t1 -> nv().visitTm(t1) ? t1 : TmPred(visitTm(t)))
        .otherwise(() -> f().TmPred(visitTm(t)))
        .visitTm(t);
  }
  default Tm TmSucc(Tm t) {
    return f().TmSucc(visitTm(t));
  }
}
```

There are three dependencies declared using abstract methods. Firstly, since Java does not support native pattern matching, the `matcher` dependency is convenient for constructing anonymous visitors. `matcher` returns an instance of the generated `TmAlgMatcher` interface, which provides fluent setters for defining visit methods via Java 8's lambdas. The `otherwise` setter mimics the wildcard pattern. Secondly, the reconstruction of a term is done via an abstract factory `f` of type `TmAlg<Tm>`. Lastly, the abstract method `nv` expresses the dependency on the visitor `Nv`.

Bool is implemented similarly in another package `bool`, whose definition is omitted. The implementation of Arith is more interesting, which is shown below:

```
@Visitor interface TmAlg<Tm> extends nat.TmAlg<Tm>, bool.TmAlg<Tm> {
  Tm TmIsZero(Tm t);
}
interface Eval1<Tm> extends GTmAlg<Tm,Tm>,bool.Eval1<Tm>,nat.Eval1<Tm> {
  TmAlgMatcher<Tm,Tm> matcher(); // Dependency refinement
  TmAlg<Tm> f();                 // Dependency refinement
  default Tm TmIsZero(Tm t) {
    return matcher()
      .TmZero(() -> f().TmTrue())
      .TmSucc(t1 -> nv(t1) ? f().TmFalse() : f().TmIsZero(visitTm(t)))
      .otherwise(() -> f().TmIsZero(visitTm(t)))
      .visitTm(t);
  }
}
interface Nv<Tm> extends TmAlgDefault<Tm,Boolean>, nat.Nv<Tm> {}
```

Nat and Bool implementations are merged using Java 8' multiple interface inheritance. Despite complementing `TmIsZero`, return types of dependencies are covariantly refined for allowing `TmIsZero` calls. Since `Nv` is implemented as a visitor, it needs to be refined as well.

Instantiation Instantiating interfaces into classes for creating objects is also required:

```
static class NvImpl implements Nv<CTm>, TmAlgVisitor<Boolean> {}
static class Eval1Impl implements Eval1<CTm>, TmAlgVisitor<CTm> {
  public TmAlg<CTm> f() { return f; }
  public TmAlgMatcher<CTm,CTm> matcher() {
    return new TmAlgMatcherImpl<>();
  }
  public Nv<CTm> nv() { return nv; }
}
static TmAlgFactory f = new TmAlgFactory();
static NvImpl nv = new NvImpl();
static Eval1Impl eval1 = new Eval1Impl();
```

The interfaces are instantiated into classes with a suffix `Impl`. `Eval1Impl`, for example, implements `Eval1` by: 1) instantiating `Tm` as the generated datatype `CTm`; 2) inheriting the generated `TmAlgVisitor` for a `visitTm` implementation; 3) fullfilling the dependencies using `TmAlgFactory`, `TmAlgMatcherImpl` and `NvImpl` respectively.

Client code The term is constructed via the factory object `f` and can be evaluated like this:

```
CTm tm = f.TmIsZero(f.TmIf(f.TmFalse(),f.TmTrue(),f.TmPred(f.TmSucc(f.TmZero()))));
eval1.visitTm(eval1.visitTm(eval1.visitTm(tm)));
```

Discussion of the approach EVF simplifies programming with visitors through code generation. It further addresses the extensibility issue by adopting extensible visitors. Restricted by Java, nested case analysis in EVF is done by means of anonymous visitors, which is not as expressive and concise as pattern matching in Scala. To enable composability, EVF visitors are defined using Java 8's interfaces with default methods—in the same spirit of using traits in Scala. Consequently, the exhaustiveness checking on the top-level visit methods is lost in visitor definition site and is delayed to the visitor instantiation site. Nevertheless, the exhaustiveness on the visit methods of the anonymous visitors is guaranteed because the `otherwise` setter must be called when constructing an anonymous visitor.

4.2.9 Castor

Highly inspired by EVF, Castor is a Scala framework designed for programming with generative, extensible visitors. Castor improves on EVF in two aspects. First, Castor adopts a more powerful Scala extensible visitor encoding presented in Section 4.2.7 that additionally enables pattern matching, GADTs, hierarchical datatypes, graphs, etc. Second, Castor employs Scalameta for annotation processing, which allows not only generating new code based on the annotated code but also modifying the annotated code itself. These extra abilities together result in more concise and expressive visitor code than that in EVF. We next give a modular implementation of Arith using Castor, which has a one-to-one correspondence with the code shown in Section 4.2.7.

Let us start with the root component `Term`:

```
@family trait Term {
  @adt trait Tm
  @default(Tm) trait Eval1 {
    type OTm = Tm
    def tm = _ => throw NoRuleApplies
  }
}
```

Several Castor's annotations are employed: `@family` denotes a Castor's component; `@adt` denotes a datatype; `@default(Tm)` denotes a default visitor on `Tm`. Compared to the `Term` definition shown in Section 4.2.7, the definition here is much simplified. The accept declaration, the type member TmV, the visitor interface TmVisit and the default visitor TmDefault are all generated by analyzing the @adt definition of Tm. Similarly, CASTOR adds the extends clause, the self type annotation and the corresponding val declaration for Eval1 by the annotation @default(Tm).

Defining Nat is also much simplified:

```
@family trait Nat extends Term {
  @adt trait Tm extends super.Tm {
    case object TmZero
    case class TmSucc(t: Tm)
    case class TmPred(t: Tm)
  }
  def nv(t: Tm): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  @default(Tm) trait Eval1 extends super.Eval1 {
    override def tmSucc = x => TmSucc(this(x.t))
    override def tmPred = {
      case TmPred(TmZero) => TmZero
      case TmPred(TmSucc(t)) if nv(t) => t
      case TmPred(t) => TmPred(this(t))
    }
  }
}
```

Variants of Tm are declared inside Tm. CASTOR will pull them outside of Tm and automatically complement the extends clause and the accept method definition. Since new variants of Tm are introduced, CASTOR will add the extended TmVisit, TmDefault and refined TmV to Nat.

Similarly, BOOL can be defined as follows:

```
@family trait Bool extends Term {
  @adt trait Tm extends super.Tm {
    case object TmTrue
    case object TmFalse
    case class TmIf(t1: Tm, t2: Tm, t3: Tm)
  }
  @default(Tm) trait Eval1 extends super.Eval1 {
    override def tmIf = {
      case TmIf(TmTrue,t2,_) => t2
      case TmIf(TmFalse,_,t3) => t3
      case TmIf(t1,t2,t3) => TmIf(this(t1),t2,t3)
    }
  }
}
```

The code below finishes the ARITH implementation:

```
@family trait Arith extends Nat with Bool {
  @adt trait Tm extends super[Nat].Tm with super[Bool].Tm {
    case class TmIsZero(t: Tm)
  }
  @visit(Tm) trait Eval1 extends super[Nat].Eval1
```

```
                         with super[Bool].Eval1 {
    def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
  }
}
```

Since the `TmIsZero` is an interesting case for `Eval1`, `@visit` annotation is used, which denotes an ordinary visitor. Thus, `Eval1` extends `TmVisit` after transformation.

Client code A `@family` trait can be directly imported in client code since Castor automatically generates a companion object for it:

```
import Arith._
val tm  = TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))
val tm1 = eval1(tm)  // iszero (pred (succ 0))
val tm2 = eval1(tm1) // iszero 0
val tm3 = eval1(tm2) // 0
```

which is identical to the client code for Scala extensible visitors shown in Section 4.2.7.

Discussion of the approach We discuss how Castor addresses the four properties:

- **Conciseness.** By employing Scala's concise syntax and metaprogramming, Castor greatly simplifies the definition and usage of visitors. In particular, the need for auxiliary visitors in performing deep case analysis is now replaced by pattern matching via `case` clauses. The concept of visitors is even made transparent to the end-user, making the framework more user-friendly.

- **Exhaustiveness.** The exhaustiveness of patterns in Castor consists of two parts. The exhaustiveness of visit methods is checked by the Scala compiler when generating companion objects. For nested patterns using `case` clauses, a default must be provided. However, this default is neither statically enforced by Scala nor Castor. Note, however, that with specialized language support it is possible to enforce that nested patterns always provide a default. This is precisely what EADDs [Zenger and Odersky, 2001] do.

- **Extensibility.** As illustrated by `Nat`, `Bool` and `Arith`, we can extend the datatype with new variants and operations, modularly. Such extensibility is enabled by the underlying extensible visitor encoding.

- **Composability.** Castor obtains composability via Scala's mixin composition, as illustrated by `Arith`. Unlike partial functions, which silently compose overlapped patterns, composing overlapped patterns in Castor will trigger compilation errors because they are conflicting methods from different traits. The error message will indicate the source of conflicts and we are free to select an implementation in resolving the conflict. The composition order does not matter as well.

Table 4.1: *Pattern matching support comparison: ●= good, ◐= neutral, ○= bad.*

	Conciseness	Exhaustiveness	Extensibility	Composability
Conventional visitors	○	●	○	○
Sealed case classes	●	●	○	○
Open case classes	●	○	●	○
Partial functions	●	○	●	◐
Extensible visitors	○	○	●	●
EVF	◐	◐	●	●
Castor	●	◐*	●	●

* Castor only gets half score on exhaustiveness because *for nested case analysis Scala cannot enforce a default*. In a language-based approach nested case analysis should always require a default, thus fully supporting exhaustiveness.

Table 4.1 summarizes the evaluation on pattern matching approaches abovementioned in terms of conciseness, exhaustiveness, extensibility, and composability. Castor is compared favorably in terms of the four properties among the approaches.

4.3 Hierarchical Datatypes

Traditional functional style datatypes are *flat*: variants have no relationships among each other. In contrast, object-oriented style datatypes (i.e. data structures modeled as class hierarchies) can be *hierarchical*: a variant can extend intermediate datatypes and/or an existing variant. In other words, while OO style class hierarchies can be arbitrarily deep, typical functional datatypes would correspond to a hierarchy where the depth is always one.

Hierarchical datatypes facilitate reuse. The subtyping relation allows the semantics defined for supertypes to be reused in subtypes. Castor supports both styles of datatypes. In this section, we illustrate Castor's support for hierarchical datatypes by revising the Arith language. Another form of hierarchical datatypes will be shown in Section 4.5, where a new variant is introduced by refining an existing variant. Moreover, the case study on UML Activity Diagrams Section 4.8 further illustrates the application of hierarchical datatypes.

4.3.1 Flat Datatypes versus Hierarchical Datatypes

Terms of the Arith language shown in Section 4.2 are represented as a flat datatype, where all the variants directly extend the root datatype `Tm`. In fact, terms can be organized in a hierarchical manner according to their types and arities. Figure 4.2 visualizes the hierarchical representation of terms and the following code materializes it using Castor:

```
@adt trait Tm {
  trait TmNullary
  trait TmUnary { val t: Tm }
  trait TmTernary { val t1, t2, t3: Tm }
  trait TmNat extends TmNullary
  trait TmBool extends TmNullary
```

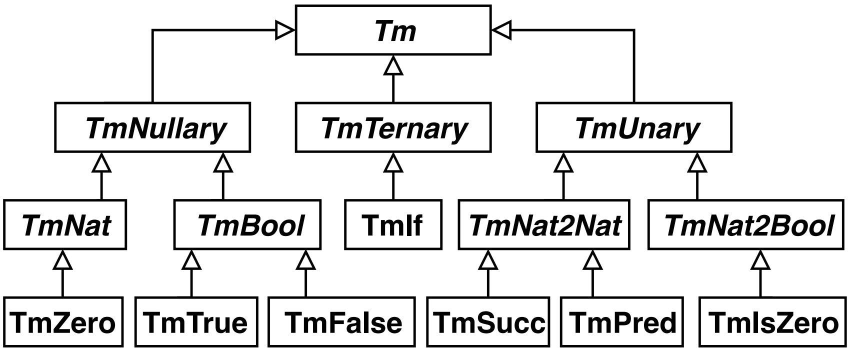


Figure 4.2: *Hierarchical representation of* Arith *terms.*

```
  trait TmNat2Nat extends TmUnary
  trait TmNat2Bool extends TmUnary
  case object TmZero extends TmNat
  case class TmSucc(t: Tm) extends TmNat2Nat
  case class TmPred(t: Tm) extends TmNat2Nat
  case object TmTrue extends TmBool
  case object TmFalse extends TmBool
  case class TmIf(t1: Tm, t2: Tm, t3: Tm) extends TmTernary
  case class TmIsZero(t: Tm) extends TmNat2Bool
}
```

The hierarchy becomes multi-layered, where several intermediate datatypes are introduced and case classes/objects do not directly extend the root but an intermediate datatype. Traits in the second layer (`TmNullary`, `TmUnary` and `TmTernary`) classify terms according to their arities. Based on arities, traits in the third layer (`TmNat`, `TmBool`, `TmNat2Nat`, `TmNat2Bool`) further classify terms according to their types. Concrete case classes/objects are in the fourth layer that extend a corresponding intermediate datatypes. For example, both `TmSucc` and `TmPred` extend `TmNat2Nat`.

4.3.2 Explicit Delegations

Now we illustrate the advantages of hierarchical datatypes. Suppose we would like to define a printer for Arith that prints out a term using an S-expression like format. For example, `TmIsZero(TmIf(TmFalse,TmTrue,TmPred(TmSucc(TmZero))))` is printed as `"(iszero (if false true (pred (succ 0))))"`. With terms being classified according to their arities, the printer can be modularized:

```
@visit(Tm) trait Print {
  type OTm = String
  def tmUnary(x: TmUnary, op: String) = "(" + op + " " + this(x.t) + ")"
  def tmSucc = tmUnary(_,"succ")
  def tmPred = tmUnary(_,"pred")
  def tmIsZero = tmUnary(_,"iszero")
  def tmZero = "0"
  def tmTrue = "true"
  def tmFalse = "false"
  def tmIf = x =>
```

```
    "(if " + this(x.t1) + " " + this(x.t2) + " " + this(x.t3) + ")"
}
```

Since all unary terms (TmSucc, TmPred and TmIsZero) are printed in the same way except for the operator, we define an auxiliary method tmUnary. Taking a TmUnary instance and an operator string as arguments, tmUnary puts the parentheses around the operator and the printed inner term of TmUnary. Then, tmSucc, tmPred and tmIsZero are implemented just by calling tmUnary with their respective instances and operator strings.

4.3.3 Default Visitors

The previous example has shown how to enhance the modularity through explicit delegations. When subtypes share the same behavior with their supertypes, the explicit delegations can be eliminated with the help of the generated default visitor. Currently, the ARITH language presented allows ill-typed terms such as TmPred(TmTrue) to be constructed. To rule out these ill-typed terms, typechecking is needed. Some of the terms share typing rules: TmTrue and TmFalse; TmSucc and TmPred. With CASTOR's default visitor, we can avoid duplication of typing rules:

```
@adt trait Ty {
  case object TyNat
  case object TyBool
}
@default(Tm) trait Typeof {
  type OTm = Option[Ty]
  override def tmBool = _ => Some(TyBool)
  override def tmNat = _ => Some(TyNat)
  override def tmNat2Nat = x => this(x.t) match {
    case Some(TyNat) => Some(TyNat)
    case _ => None
  }
  override def tmNat2Bool = x => this(x.t) match {
    case Some(TyNat) => Some(TyBool)
    case _ => None
  }
  override def tmIf = x => (this(x.t1),this(x.t2),this(x.t3)) match {
    case (Some(TyBool),ty1,ty2) if ty1 == ty2 => this(x.t2)
    case _ => None
  }
  def tm = _ => None
}
```

Like Tm, Ty is a datatype for representing types, where TyNat and TyBool are two concrete types. A visitor Typeof is defined for typechecking terms. The output type of Typeof is Option[Ty], indicating that if a term is well-typed, some type will be returned; otherwise a None will be returned. Except for TmIf, typing rules are defined on intermediate datatypes. For example, tmNat2Nat is overridden, which checks whether its inner term is of type TyNat and returns TyNat if so. tmSucc and tmPred are implicitly implemented by the inherited default visitor, whose definition is given below:

```
trait TmDefault extends TmVisit { _: TmV =>
```

```
  def tm: Tm => OTm
  def tmNullary = (x: TmNullary) => tm(x)
  def tmUnary = (x: TmUnary) => tm(x)
  def tmTernary = (x: TmTernary) => tm(x)
  def tmNat = (x: TmNat) => tmNullary(x)
  def tmBool = (x: TmBool) => tmNullary(x)
  def tmNat2Nat = (x: TmNat2Nat) => tmUnary(x)
  def tmNat2Bool = (x: TmNat2Bool) => tmUnary(x)
  def tmZero = tmNat(TmZero)
  def tmSucc = tmNat2Nat(_)
  def tmPred = tmNat2Nat(_)
  def tmTrue = tmBool(TmTrue)
  def tmFalse = tmBool(TmFalse)
  def tmIf = tmTernary(_)
  def tmIsZero = tmNat2Bool(_)
}
```

We can see that the default visitor extends the visitor interface with visit methods for intermediate datatypes and each visit method is implemented by delegating to its direct parent's visit method.

4.4 GADTs and Well-Typed EDSLs

In this section, we show the support for *generalized algebraic data types* (GADTs) [Xi et al., 2003] in Castor. GADTs allow not only datatypes to be parameterized but also well-formedness constraints to be expressed in constructors. GADTs are widely used for building well-typed domain-specific languages (EDSLs), which exploit the type system of the host language to typecheck the terms of the EDSL. Popular approaches to EDSLs like Finally Tagless [Carette et al., 2009] can provide an encoding of GADTs and provide modularity as well. However, the encoding employed by Finally Tagless is based on Church encodings. Unfortunately, this makes it hard to model several operations that require nested patterns or operations with dependencies. The interested reader is referred to Section 2 and 3 of the EVF paper [Zhang and Oliveira, 2017] for a detailed discussion on the issue of Church encodings. We show that just as Finally Tagless encodings, modularity is supported, and like GADTs nested pattern matching and dependencies are easy to do as well.

4.4.1 GADTs and Well-Typed Terms

We have shown how to rule out ill-typed terms using a type-checking algorithm in Section 4.3.3. A better solution, however, is to prevent such terms from being constructed in the first place. This is possible through representing Arith terms using a GADT-style:

```
@family trait GArith {
  @adt trait Tm[A] {
    case object TmZero extends Tm[Int]
    case class TmSucc(t: Tm[Int]) extends Tm[Int]
    case class TmPred(t: Tm[Int]) extends Tm[Int]
    case object TmTrue extends Tm[Boolean]
    case object TmFalse extends Tm[Boolean]
    case class TmIf[A](t1: Tm[Boolean], t2: Tm[A], t3: Tm[A])
      extends Tm[A]
    case class TmIsZero(t: Tm[Int]) extends Tm[Boolean]
  }
}
```

`Tm` is now parameterized by a type parameter `A`. When declaring variants of `Tm`, the `extends` clause cannot be omitted anymore since Castor does not know how to instantiate `A`. Notice that `A` is instantiated differently as `Int` or `Boolean` for expressing well-formedness constraints. For example, `TmIsZero` requires its subterm `t` of type `Tm[Int]`. Consequently, one cannot supply a term of type `Tm[Boolean]` constructed from `TmTrue`, `TmFalse` or `TmIsZero` to `TmIsZero`. Therefore, ill-formed terms are statically rejected by the Scala type system:

```
TmIsZero(TmZero) // Accepted!
TmIsZero(TmTrue) // Rejected!
```

4.4.2 Well-Typed Big-Step Evaluator

As opposed to small-step semantics, big-step semantics immediately evaluates a valid term to a value. In the case of Arith, a term can either be evaluated to an integer or a boolean value. Without GADTs, implementing a big-step evaluator for Arith is tedious:

```
@family @adts(Tm) @ops(Eval1) trait EvalArith extends Arith {
  @adt trait Value {
    case class IntValue(v: Int)
    case class BoolValue(v: Boolean)
  }
  @visit(Tm) trait Eval {
    type OTm = Value
    def tmZero = IntValue(0)
    def tmSucc = x => this(x.t) match {
      case IntValue(n) => IntValue(n+1)
      case _ => throw NoRuleApplies
    }
    def tmPred = x => this(x.t) match {
      case IntValue(n) => IntValue(n-1)
      case _ => throw NoRuleApplies
    }
    def tmTrue = BoolValue(true)
    def tmFalse = BoolValue(false)
    def tmIf = x => this(x.t1) match {
      case BoolValue(true) => this(x.t2)
      case BoolValue(false) => this(x.t3)
      case _ => throw NoRuleApplies
    }
    def tmIsZero = x => this(x.t) match {
      case IntValue(0) => BoolValue(true)
      case IntValue(_) => BoolValue(false)
      case _ => throw NoRuleApplies
    }
  }
}
```

`EvalArith` illustrates the operation extensibility of Castor, which does not introduce any new variants of `Tm` but a new visitor `Eval` on `Tm`. Auxiliary annotations **`@adts`** and **`@ops`** provide inherited datatypes and operations for Castor to generate the companion object. Such an implementation suffers from the so-called *tag* problem [Carette et al., 2009]: to accommodate different evaluation result types, an open datatype `Value` is defined for accommodating integers, booleans and many other evaluation result types that might be added in the future. The two variants `IntValue` and `BoolValue` are introduced for wrapping integers and boolean values, respectively. Pattern matching is used for unwrapping the evaluation results from inner terms. A defensive wildcard is needed for dealing with ill-typed terms. We can see that the tagging overhead is high.

Fortunately, we can avoid the tag problem with the help of Castor's GADTs. The extensible visitor encoding for GADTs is slightly different from the one presented in Section 4.2.7, which additionally take the type information carried by terms into account. For instance, the visitor interface generated for `Tm[A]` is listed below:

```
trait TmVisit { _: TmV =>
  type OTm[A]
  def apply[A](x: Tm[A]) = x.accept(this)
  def tmZero: OTm[Int]
  def tmSucc: TmSucc => OTm[Int]
  def tmPred: TmPred => OTm[Int]
  def tmTrue: OTm[Boolean]
  def tmFalse: OTm[Boolean]
  def tmIf[A]: TmIf[A] => OTm[A]
  def tmIsZero: TmIsZero => OTm[Boolean]
}
```

Each visit method now returns a value of a higher-kinded type `OTm[A]`, where `A` is instantiated consistently with how it is instantiated in the **`extends`** clause. For example, `tmZero` is of type `OTm[Int]` while `tmTrue` is of type `OTm[Boolean]`. Then, a well-typed big-step evaluator can be made *tagless*:

```
@family @adts(Tm) trait EvalGArith extends GArith {
  @visit(Tm) trait Eval {
    type OTm[A] = A
    def tmZero = 0
    def tmSucc = x => this(x.t) + 1
    def tmPred = x => this(x.t) - 1
    def tmTrue = true
    def tmFalse = false
    def tmIf[A] = x => if (this(x.t1)) this(x.t2) else this(x.t3)
    def tmIsZero = x => this(x.t) == 0
  }
}
```

With the output type specified as `A`, the visit method returns a value of the type carried by the term. For example, visit methods `tmZero` and `tmTrue` return `Int` and `Boolean` values respectively. Moreover, this `Eval` implementation remains retroactive when terms of new types (such as `Tm[Float]`) are introduced.

Here are some terms that have different evaluation result types.

```
import EvalGArith._
eval(TmSucc(TmZero))   // 1
eval(TmIsZero(TmZero)) // true
```

4.4.3 Well-Typed Small-Step Evaluator

Well-typed big-step evaluators can be defined with Finally Tagless in an equally simple manner. What distinguishes Castor from Finally Tagless is the ability to define small-step evaluators in an easy way. The need for deep patterns and the dependency on a numeric value checker causes immediate trouble for Finally Tagless. Although workarounds may be possible for some of the issues, they are cumbersome and require significant amounts of boilerplate code [Kiselyov, 2012]. In contrast, encoding small-step semantics in a GADT-style with Castor is unproblematic:

```
@family @adts(Tm) trait Eval1Arith extends GArith {
  def nv[A](t: Tm[A]): Boolean = t match {
    case TmZero => true
    case TmSucc(t1) => nv(t1)
    case _ => false
  }
  @default(Tm) trait Eval1 {
    type OTm[A] = Tm[A]
    def tm[A] = x => throw NoRuleApplies
    override def tmIf[A] = {
      case TmIf(TmTrue,t2,_) => t2
      case TmIf(TmFalse,_,t3) => t3
      case TmIf(t1,t2,t3) => TmIf(this(t1),t2,t3)
    }
    override def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
    ... // Other cases are the same as before
  }
}
```

The instantiation of the output type guarantees that the small-step evaluator is *type-preserving*. That is, the type carried by a term remains the same after one step of evaluation. For example, calling `eval1` on `TmZero` will never return `TmTrue` no matter how `Eval1` is implemented. The actual definition of `Eval1` is almost the same as before except that `nv`, `tm` and `tmIf` become generic. Still, the ability to do nested pattern matching and to call `nv` in `Eval1` is preserved.

4.4.4 Extension: Higher-Order Abstract Syntax for Name Binding

A recurring problem in designing EDSLs is how to deal with binders. For example, in lambda calculus, operations involved with names like α-equivalence and capture-avoiding substitution are non-trivial to define. Higher-order abstract syntax (HOAS) [Pfenning and Elliott, 1988] avoids these problems through reusing the binding mechanisms provided by the host language. The following code shows how to extend Arith with simply-typed lambda calculus modularly:

```
@family trait HOAS extends EvalGArith {
  @adt trait Tm[A] extends super.Tm[A] {
    case class TmVar[A](v: A) extends Tm[A]
    case class TmAbs[A,B](f: Tm[A] => Tm[B]) extends Tm[A => B]
    case class TmApp[A,B](t1: Tm[A => B], t2: Tm[A]) extends Tm[B]
  }
  @visit(Tm) trait Eval extends super.Eval {
    def tmVar[A] = _.v
    def tmAbs[A,B] = x => y => this(x.f(TmVar(y)))
    def tmApp[A,B] = x => this(x.t1)(this(x.t2))
  }
}
```

Three new forms of terms are introduced: lifters (`TmVar`), lambda abstractions (`TmAbs`) and applications (`TmApp`). Of particular interest is `TmAbs`, which constructs a term of type `Tm[A => B]` from a Scala lambda function `Tm[A] => Tm[B]` and thus is *higher-order*.

Correspondingly, `Eval` is extended with three new visit method implementations. `tmVar` simply extracts the value out of the lifter. `tmAbs` is trickier since it returns a value of type `A => B`. A lambda function is hence created, which takes `y` of type `A` and lifts it into `Tm[A]` using `TmVar`, then applies `x.f` to the lifted term for computing a `Tm[B]` and finally does a recursive call to evaluate `Tm[B]` into `B`. `tmApp` recursively evaluates `t1` and `t2`, which returns the value `A => B` and `A` respectively. Then it applies `A => B` to `A` for getting a value of `B`.

Here is an example that illustrates the use of `HOAS`:

```
import HOAS._
eval(TmApp(TmAbs((t: Tm[Int]) => TmSucc(TmSucc(t))), TmZero)) // 2
```

We first create an abstraction term that applies successor twice to the argument `t` and then apply it to constant zero. Note that the type of `t` is explicitly specified because Scala's type system is not powerful enough to infer the type of `TmAbs` without the type annotation.

4.5 Graphs and Imperative Visitors

Examples presented so far are all *functional* visitors (i.e. computation is done via returning values) on immutable trees. In fact, Castor also supports *imperative* visitors (i.e. computation is done via side effects) and the data structure can be a mutable graph. Imperative computation is, in some cases, more efficient than the functional counterpart regarding time and memory. Compared to trees, graphs are a more general data structure that have many important applications. For instance, in the domain of compilers, abstract semantic graphs can be used for representing shared subexpressions, facilitating optimizations like common subexpression elimination. In this section, we show how to model graphs and imperative visitors with Castor.

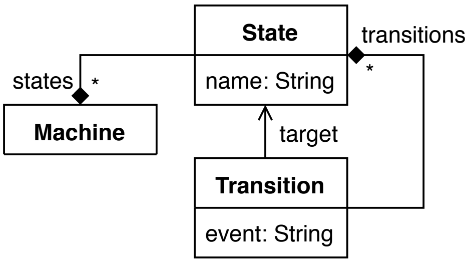


Figure 4.3: *Class diagram of FSM.*

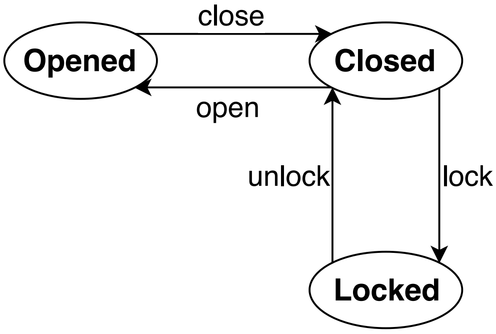


Figure 4.4: *A state machine for controlling a door.*

4.5.1 The Difficulties in Modeling Graphs

Modeling graphs modularly is non-trivial in approaches such as Object Algebras. Consider modeling a Finite State Machine (FSM) language. Figure 4.3 shows a UML class diagram for the FSM language. A `Machine` consists of some `State`s. Each `State` has a name and a number of `Transition`s. A `Transition` is triggered by an `event`, taking one `State` to another. Concretely, Figure 4.4 shows a simple state machine for controlling a door, which has three states (opened, closed and locked) and four transitions (close, open, unlock and lock). From Figure 4.4 we can see that this state machine forms a *graph*, where we can go back and forth from one state to another along with the transitions.

A failed attempt with Object Algebras Let us try to model the FSM language with Object Algebras [Oliveira and Cook, 2012]. Describing the FSM language using an Object Algebra interface is unproblematic:

```
trait FSM[M,S,T] {
  def machine(states: List[S]): M
  def state(name: String, trans: List[T]): S
  def trans(event: String, target: S): T
}
```

where `M`, `S`, `T` and their variants are captured as type parameters and factory methods respectively. However, constructing a graph using this representation is hard because Object Algebras support only immutable tree structures that are built *bottom up*. Here is a failed attempt on modeling the door state machine:

```
// Forward reference error!
def door[M,S,T](f: FSM[M,S,T]) = {
  val close: T = f.trans("close",closed)
  val open: T = f.trans("open",opened)
  val lock: T = f.trans("lock",locked)
  val unlock: T = f.trans("unlock",closed)
  val opened: S = f.state("opened", List(close))
  val closed: S = f.state("closed", List(open,lock))
  val locked: S = f.state("locked", List(unlock))
  f.machine(List(opened,closed,locked))
}
```

A *forward reference* error will always occur no matter how we arrange these statements. The reason is that there is no proper way to decouple the cyclic references between states and transitions.

4.5.2 FSM in Castor

Fortunately, modeling the FSM language using Castor is not a problem:

```
@family trait FSM {
  @adt trait M {
    val states = ListBuffer[S]()
    class Machine
  }
  @adt trait S {
    val trans = ListBuffer[T]()
    var name: String
    class State(var name: String)
  }
  @adt trait T {
    class Trans(val event: String, var target: S)
  }
  @visit(M,S,T) trait Print {
    type OM = String
    type OS = OM
    type OT = OM
    def machine = _.states.map{this(_)}.mkString("\n")
    def state = s => s.trans.map{this(_)}.mkString(s.name+":\n","\n","")
    def trans = t => t.event + " -> " + t.target.name
  }
  @visit(M,S,T) trait Step {
    type OM = String => Unit
    type OS = OM
    type OT = OM
    var res: S = null
    def machine = m => event => m.states.foreach{this(_)(event)}
    def state = s => event => s.trans.foreach{this(_)(event)}
    def trans = t => event => if (event == t.event) res = t.target
  }
}
```

The actual class hierarchies of the FSM language are slightly different from what Figure 4.3 shows. Each class in the UML diagram is defined inside an **@adt** trait for allowing potential variant extensions. Fields are either declared as **var** or **val** for enabling/disabling mutability.

Combined visitors There are two visitors defined for the FSM language, namely `Print` and `Step`. Both of them are *combined* visitors that apply to transitions, states, and machines. Such a combined implementation is much more compact than defining three mutually dependent visitors with distinct names. Annotated as `@visit(M,S,T)`, `Print` instantiates the output types `OM`, `OS`, `OT` consistently as `String` and implements three visit methods `machine`, `state` and `trans` altogether. Concretely, methods `machine` and `state` map `Print` to the substructures and concatenate the results with a newline. For `trans`, we should not call `this` on the `target` state otherwise it will not terminate. Instead, we print out the `name` field on the `target` state only.

Imperative visitors The `Step` visitor captures the small-step execution semantics of FSM. Given an event, it goes through the structure for finding out the transition triggered by that event and returning the state that transition points to. Note that `Step` is, at the same time, an *imperative* visitor. `Step` instantiates the output types as `String => Unit` and updates the field `res` to the found target transition. If `res` is still `null` after traversal, then no such transition exists.

Now we are able to model the state machine that controls doors like this:

```
import FSM._

val door = new Machine
val opened = new State("Opened")
val closed = new State("Closed")
val locked = new State("Locked")
val open = new Trans("open",opened)
val close = new Trans("close",closed)
val lock = new Trans("lock",locked)
val unlock = new Trans("unlock",closed)

door.states += (opened,closed,locked)
opened.trans += close
closed.trans += (open,lock)
locked.trans += unlock
```

The graph is constructed in a conventional OOP style. Unlike Object Algebras, the structure is built *top down*. To decouple cyclic references, the declaration and initialization of the variables are separated. This is possible in Castor because variants are concrete classes provided with setters whereas in Object Algebras they are abstract types without concrete representations.

Calling `print(door)` produces the following output:

```
Opened:
close -> Closed
Closed:
open -> Opened
lock -> Locked
Locked:
unlock -> Closed
```

Some tests on `Step` are:

```
step(door)("open")
println(step.res.name) // "Opened"
step.res = null        // Reset to null
step(door)("close")
println(step.res.name) // "Closed"
```

Imperative visitors should be used more carefully. In the case of Step, its field res needs to be reset to null afterwards. Otherwise, the result may be wrong next time we call step.

4.5.3 Language Composition and Memoized Traversals

Consider unifying FSM and Arith. The unification happens when a new kind of transition called guarded transitions is introduced. A guarded transition additionally contains a boolean term and is triggered not only by the event but also by the evaluation result of that term. Combining FSM with the GADT version of Arith is given below:

```
@family @adts(Tm,F,S) @ops(Eval)
trait GuardedFSM extends FSM with EvalArith {
  @adt trait T extends super[FSM].T {
    class GuardedTrans(event: String, target: State, val tm: Tm[Boolean])
      extends Trans(event, target)
  }
  @visit(M,S,T) trait Print extends super[FSM].Print {
    def guardedTrans = t => trans(t) + " when " + t.tm.toString
  }
  @visit(F,S,T) trait Step extends super[FSM].Step {
    def guardedTrans = t => event => if (eval(t.tm)) trans(t)(event)
  }

  @visit(S,T) trait Reachable {
    type OS = Unit
    type OT = Unit
    val reached = collection.mutable.Set[S]()
    def state = s =>
      if (!reached.contains(s)) {
        reached += s
        s.trans.foreach(this(_))
      }
    def trans = t => this(t.target)
    def guardedTrans = t => if (eval(t.tm)) this(t.target)
  }
}
```

Class GuardedTrans extends Trans with an additional field tm of type Tm[Boolean]. To handle GuardedTrans, Print and Step are extended with an implementation of guardedTrans method. Having GuardedTrans as a subtype of Trans, we are able to partially reuse the semantics of Trans for GuardedTrans via passing t to the inherited trans method.

Memoized traversals Naively traversing a graph might be inefficient because the same object may be traversed multiple times. In the worst case, the traversal may not even terminate if not dealt with carefully. A better approach is to memoize the results of traversed objects and fetch the cached result when an object is traversed again. Reachable

$$
\begin{array}{lcl}
\mathit{Fam} & ::= & \texttt{@family @adts}(\overline{D})\ \texttt{@ops}(\overline{V})\ \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\ \{\overline{\mathit{Adt}}\ \overline{\mathit{Vis}}\} \\
\mathit{Adt} & ::= & \texttt{@adt trait}\ D[\overline{X}]\ \texttt{extends}\ \overline{\texttt{super}[F].D[\overline{X}]}\ \{\overline{\mathit{Ctr}}\} \\
\mathit{Ctr} & ::= & \texttt{class}\ C[\overline{X}]\ \texttt{extends}\ (C[\overline{T}]\ \texttt{with})?\ \overline{D[\overline{T}]} \\
 & | & \texttt{object}\ C\ \texttt{extends}\ (C[\overline{T}]\ \texttt{with})?\ \overline{D[\overline{T}]} \\
 & | & \texttt{trait}\ D[\overline{X}]\ \texttt{extends}\ \overline{D[\overline{T}]} \\
\mathit{Vis} & ::= & \texttt{@}(\texttt{default} \mid \texttt{visit})(\overline{D})\ \texttt{trait}\ V\ \texttt{extends}\ \overline{\texttt{super}[F].V} \\
T & ::= & X \mid D[\overline{T}] \mid \texttt{Int} \mid T \texttt{=>} T
\end{array}
$$

Figure 4.5: Castor *syntax.*

is a combined imperative visitor that finds out all reachable states for the given state. The reachable states are collected in a `reached` field, which is initialized as an empty mutable set. `Reachable` employs memoized depth-first search, which first checks whether the state has already been traversed. If not, the state is added to `reached` and the recursion goes to the states its transitions lead to. Similarly, memoization can be applied to functional visitors by changing `reached` to a mutable map.

We can build a guarded door controller by changing the **`import`** statement and how `lock` is initialized:

```
val lock = new GuardedTrans("lock",locked,TmFalse)
```

Now, an opened door can no longer be locked because the guard evaluates to `false`:

```
reachable(open)
println(reachable.reached.size) // 2
```

By setting the expression to `TmTrue`, the door can be locked again:

```
lock.tm = TmTrue
reachable.clear // Reset to empty
reachable(open)
println(reachable.reached.size) // 3
```

4.6 Formalized Code Generation

In previous sections, we have shown code written with Castor and its corresponding generated code. In this section, we formally describe the valid Scala programs accepted by Castor and the transformation scheme.

4.6.1 Syntax

Figure 5.1 describes valid Scala programs accepted by Castor. Uppercase metavariables range over capitalized names. $\overline{A}$ is written as a shorthand for a potentially empty sequence $A_1 \bullet \ldots \bullet A_n$, where $\bullet$ denotes **`with`**, comma or semicolon depending on the context. $(\ldots)?$ denotes that $\ldots$ is optional. For brevity, we ignore the syntax that is irrelevant to the transformation, such as the **`case`** modifier, constructors, fields, and methods. These parts are kept unchanged after transformation.

4.6.2 Transformation

Figure 4.6 formalizes the transformation. We use semantic brackets ($\llbracket\cdot\rrbracket$) in defining the transformation rules and angle brackets (<>) for processing sequences. The transformation is given by pattern matching on the concrete syntax and is quite straightforward. One can see that processing the Arith implementation in Castor (cf. Section 5.2) through Figure 4.6 will get back the extensible visitor implementation (cf. Section 4.2.7).

Here we briefly discuss some interesting cases. A `trait` is recognized as a *base case* if it extends nothing. Base cases have extra declarations such as `accept` declaration for datatypes or `val` declaration for visitors. Variants declared using `class`, `trait` or `object` are treated differently. `object`s and `class`es have their corresponding visit methods in the visitor interface while visit methods for `trait`s only exist in the default visitor. The `extends` clause for `@adt` is used in inferring the `extends` clause for concrete visitors.

4.6.3 Implementation

Castor employs Scalameta [Burmako, 2017] (version 1.8.0), a modern Scala metaprogramming library, for analyzing and generating the code. The actual implementation closely follows the formalization. After parsing, the Scala source program is represented as an AST. We first check the validity of that AST with errors like annotating `@adt` not on a trait reported. We then generate code by analyzing the AST. Next, we build the AST with code injected. Finally, the AST is typechecked by the Scala compiler. During the process, Scala's quasiquotes are used, which allow us to analyze and rebuild the AST conveniently via the concrete syntax.

4.7 Case Study I: Types and Programming Languages

In this section, we present a case study on modularizing the interpreters in TAPL [Pierce, 2002]. The Arith language and its variations are directly from or greatly inspired by the TAPL case study. TAPL are a good benchmark for examining Castor's capabilities of open pattern matching and modular dependencies. The reason is that core data structures of TAPL interpreters, types and terms, are modeled using algebraic datatypes; operations over types and terms are defined via pattern matching. There are a few operations that require nested patterns: small-step semantics, type equality, and subtyping relations. They all come with a default. The data structures and associated operations should be modular as new language features are introduced and combined. However, without proper support for modular pattern matching, the original implementation duplicates code for features that could be shared. With Castor and techniques shown in Section 5.2, we are able to refactor the non-modular implementation into a modular manner. Our evaluation shows that the refactored version significantly reduces the SLOC compared to a non-modular implementation found online. However, at the moment, improved modularity does come at some performance penalty.

$$
\begin{array}{l}
\llbracket \texttt{@family @adts}(\overline{D})\ \texttt{@ops}(\overline{V})\ \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\ \{\overline{Adt}\ \overline{Vis}\}\rrbracket =\\
\quad \texttt{trait}\ F\ \texttt{extends}\ \overline{F}\ \{\llbracket\overline{Adt}\rrbracket\ \llbracket\overline{Vis}\rrbracket\}\\
\quad \texttt{object}\ F\ \texttt{extends}\ F\ \{\\
\qquad \langle \texttt{type}\ D\texttt{V = }D\texttt{Visit} \mid D \in \overline{D} \cup \overline{Adt}\rangle\\
\qquad \langle \texttt{object}\ \upsilon\ \texttt{extends}\ V \mid V \in \overline{V}\cup\overline{Vis}\rangle\\
\quad \}\\
\llbracket \texttt{@adt trait}\ D[\overline{X}]\ \{\overline{Ctr}\}\rrbracket =\\
\quad \texttt{type}\ D\texttt{V <:}D\texttt{Visit}\\
\quad \texttt{trait}\ D[\overline{X}]\ \{\texttt{def accept(v:}D\texttt{V): v.O}D[\overline{X}]\}\\
\quad \llbracket\overline{Ctr}\rrbracket\\
\quad \texttt{trait}\ D\texttt{Visit}\ \{\ _\texttt{:}D\texttt{V =>}\\
\qquad \texttt{type O}D[\overline{X}]\\
\qquad \texttt{def apply}[\overline{X}]\texttt{(x:}D[\overline{X}]\texttt{) = x.accept(this)}\\
\qquad \llbracket\overline{Ctr}\rrbracket_{visit}\\
\quad \}\\
\quad \texttt{trait}\ D\texttt{Default extends}\ D\texttt{Visit}\ \{\ _\texttt{:}D\texttt{V =>}\\
\qquad \texttt{def}\ d[\overline{X}]\texttt{:}D[\overline{X}]\ \texttt{=> O}D[\overline{X}]\\
\qquad \llbracket\overline{Ctr}\rrbracket_{default}\\
\quad \}\\
\llbracket \texttt{@adt trait}\ D\ \texttt{extends}\ \overline{\texttt{super}[F]\texttt{.}D}\ \{\overline{Ctr}\}\rrbracket =\\
\quad \texttt{type}\ D\texttt{V <:}D\texttt{Visit}\\
\quad \llbracket\overline{Ctr}\rrbracket\\
\quad \texttt{trait}\ D\texttt{Visit extends}\ \overline{\texttt{super}[F]\texttt{.}D\texttt{Visit}}\ \{\ _\texttt{:}D\texttt{V =>}\llbracket\overline{Ctr}\rrbracket_{visit}\}\\
\quad \texttt{trait}\ D\texttt{Default extends}\ D\texttt{Visit with}\ \overline{\texttt{super}[F]\texttt{.}D\texttt{Default}}\ \{_\texttt{:}D\texttt{V =>}\ \llbracket\overline{Ctr}\rrbracket_{default}\}\\
\llbracket\texttt{class}\ C[\overline{X}]\ \ldots\rrbracket = \texttt{class}\ C[\overline{X}]\ \ldots\ \{\texttt{override def accept(v:}D\texttt{V) = v.}c\texttt{(this)}\}\\
\llbracket\texttt{object}\ C\ \ldots\rrbracket = \texttt{object}\ C\ \ldots\ \{\texttt{override def accept(v:}D\texttt{V) = v.}c\}\\
\llbracket Ctr\rrbracket = Ctr\\
\llbracket\texttt{class}\ C[\overline{X}]\ \texttt{extends}\ (\ldots\texttt{with})?\ D[\overline{T}]\rrbracket_{visit} = \texttt{def}\ c[\overline{X}]\texttt{:}C\ \texttt{=> O}D[\overline{T}]\\
\llbracket\texttt{object}\ C\ \texttt{extends}\ (\ldots\ \texttt{with})?\ D[\overline{T}]\rrbracket_{visit} = \texttt{def}\ c\texttt{: O}D[\overline{T}]\\
\llbracket Ctr\rrbracket_{visit} = \varnothing\\
\llbracket\texttt{class}\ C_1[\overline{X}]\ \texttt{extends}\ C_2[\overline{T}]\ldots\rrbracket_{default} = \texttt{def}\ c_1[\overline{X}]\texttt{= x =>}\ c_2\texttt{(x)}\\
\llbracket\texttt{object}\ C_1\ \texttt{extends}\ C_2[\overline{T}]\ldots\rrbracket_{default} = \texttt{def}\ c_1\ \texttt{=}\ c_2\texttt{(}C_1\texttt{)}\\
\llbracket\texttt{trait}\ D_1[\overline{X}]\ \texttt{extends}\ D_2[\overline{T}]\ldots\rrbracket_{default} = \texttt{def}\ d_1\ \texttt{= (x:}D_1[\overline{X}]\texttt{) =>}\ d_2\texttt{(x)}\\
\llbracket\texttt{@}(\texttt{default}\mid\texttt{visit})(\overline{D})\ \texttt{trait}\ V\rrbracket =\\
\quad \texttt{trait}\ V\ \texttt{extends}\ \overline{D(\texttt{Default}\mid\texttt{Visit})}\ \{\ _\texttt{:}\overline{D\texttt{V}}\texttt{=>}\ldots\}\\
\quad \texttt{val}\ \upsilon : V\\
\llbracket\texttt{@}(\texttt{default}\mid\texttt{visit})(\overline{D})\ \texttt{trait}\ V\ \texttt{extends}\ \overline{\texttt{super}[F]\texttt{.}V}\rrbracket =\\
\quad \texttt{trait}\ V\ \texttt{extends}\ \overline{D(\texttt{Default}\mid\texttt{Visit})}\ \texttt{with}\ \overline{\texttt{super}[F]\texttt{.}V}\ \{\ _\texttt{:}\overline{D\texttt{V}}\texttt{=>}\ldots\}\\
\llbracket\overline{X}\rrbracket = \langle\llbracket X\rrbracket \mid X\in\overline{X}\rangle
\end{array}
$$

Figure 4.6: Castor *transformation.*

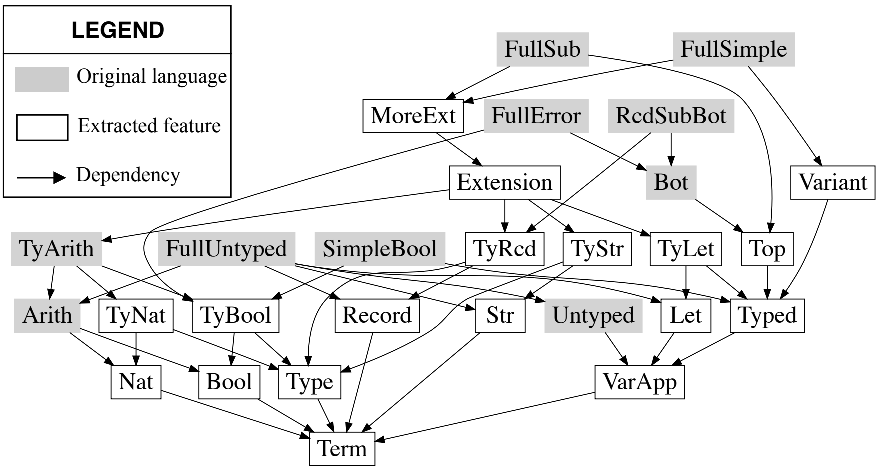


Figure 4.7: *Simplified language/feature dependency graph.*

4.7.1 Overview

An existing Scala implementation of TAPL[2] strictly follows the original OCaml version, which uses sealed case classes and pattern matching. The first ten languages (*arith*, *untyped*, *fulluntyped*, *tyarith*, *simplebool*, *fullsimple*, *fullerror*, *bot*, *rcdsubbot* and *fullsub*) are our candidates for refactoring. Each language implementation consists of 4 files: *parser*, *syntax*, *core* and *demo*. These languages cover various features including arithmetic, lambda calculus, records, fixpoints, error handling, subtyping, etc. Features are shared among these ten languages. However, such featuring sharing is achieved via duplicating code, causing problems like:

- **Inconsistent definitions.** Lambdas are printed as `"lambda"` in all languages except for *untyped*, where lambdas are printed as `"\"`.
- **Feature leaks.** Features introduced in the latter part of the book (e.g., System F) leak to previous language implementations such as *fullsimple*.

Our refactoring focuses on *syntax* and *core* where datatypes and associated operations are defined. Figure 4.7 gives a simplified high-level overview of the refactored implementation. The candidate languages are represented as gray boxes whereas extracted features/sub-languages are represented as white boxes. From Figure 4.7 we can see that the interactions between languages (revealed by the arrows) are quite intense. Take ARITH for example, it is a sublanguage for *tyarith*, *fulluntyped*, *fullerror*, *fullsimple* and *fullsub*. Unfortunately, without proper modularization techniques, the original implementation repeats the definition of *arith* at least five times. In the refactored implementation written with CASTOR, however, *arith* is defined only once and modularly reused in other places.

4.7.2 Evaluation

We evaluate CASTOR by answering the following questions:

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

Table 4.2: *SLOC evaluation of TAPL interpreters*

Extracted	Castor	**EVF**	**Language**	Castor	**EVF**	**Scala**
bool	71	98	arith	31	33	106
extension	24	34	untyped	40	46	124
str	42	55	fulluntyped	18	47	256
let	48	47	tyarith	22	26	157
moreext	112	106	simplebool	24	38	212
nat	85	103	fullsimple	24	83	619
record	117	198	fullerror	68	105	396
top	79	86	bot	40	61	190
typed	82	138	rcdsubbot	30	39	257
varapp	40	65	fullsub	57	116	618
variant	136	161				
misc	212	172	**Total**	**1402**	**1857**	**2935**

- **Q1.** Is Castor effective in reducing SLOC?
- **Q2.** How does Castor compare to EVF?
- **Q3.** How much performance penalty does Castor incur?

Q1 Table 4.2 reports the SLOC comparison results. With all the features/sublanguages extracted, implementing a candidate language with Castor is merely done by composing features/sublanguages. Therefore, the more features/sublanguages the candidate language uses, the more code Castor reduces. Compared to the non-modular Scala implementation, for a simple language like *arith*, the reduction rate[3] is 71%; for a feature-rich language like *fullsimple*, the reduction rate can be up to 96%. Overall, Castor reduces over *half* of the total SLOC with respect to the non-modular version.

Q2 Table 4.2 also compares Castor with EVF [Zhang and Oliveira, 2017]. Castor reduces over 400 SLOC compared to EVF. As we have shown in Section 4.2, the reduction comes from the native support for pattern matching, generated dependency declarations, etc. More importantly, the instantiation burden for EVF is heavy if there are a lot of visitors and the dependencies are complex. In contrast, Castor completely removes the instantiation burden by generating companion objects automatically.

Q3 To measure the performance, we randomly generate 10,000 terms for each language and calculate the average evaluation time for 10 runs. The ScalaMeter[4] microbenchmark framework is used for performance measurements. The benchmark programs are compiled using Scala 2.12.7, JDK version 1.8.0_211 and are executed on a MacBook Pro with 2.3 GHz quad-core Intel Core i5 processor with 8 GB memory. Figure 4.8 compares the execution time in milliseconds. From the figure we can see that Castor implementations

[3] Reduction rate = $\frac{\textbf{Scala}\ \text{SLOC} - \text{Castor SLOC}}{\textbf{Scala}\ \text{SLOC}} \times 100\%$

[4] http://scalameter.github.io

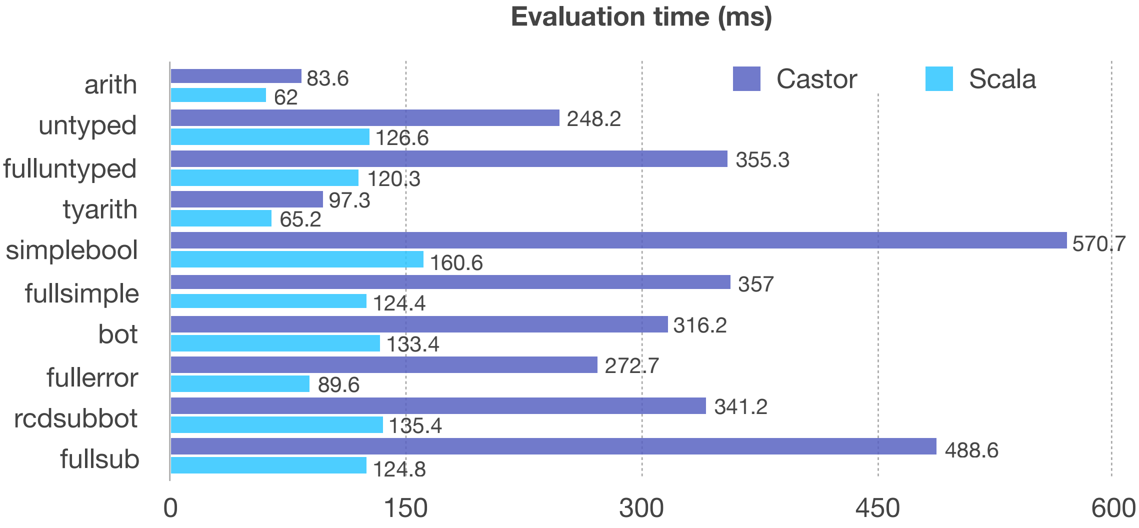


Figure 4.8: *Performance evaluation of TAPL interpreters.*

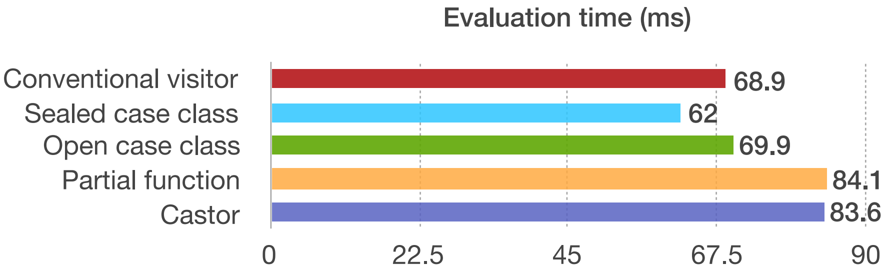


Figure 4.9: *Performance evaluation of* Arith.

have a 1.35x (*arith*) to 3.92x (*fullsub*) slowdown with respect to the corresponding non-modular Scala implementations. The more features a modular implementation combines, the more significant the slowdown is. Figure 4.9 further compares the performance of the Scala Arith implementations discussed in Section 4.2. Obviously, modular implementations are slower than non-modular implementations. With the underlying optimizations, the implementation based on sealed case classes is faster than the implementation based on conventional visitors.

We believe that the performance penalty is mainly caused by method dispatching. A modular implementation typically has a complex inheritance hierarchy. Dispatching on a case needs to go across that hierarchy. Thus, the more complex the hierarchy is, the worse the performance is. Another source of performance penalty might be the use of functions instead of normal methods in visitors. Of course, more rigorous benchmarks need to be conducted to verify our guesses. One possible way to boost the performance is to turn TAPL interpreters into compilers via staging using the LMS framework [Rompf and Odersky, 2010]. This is currently not possible because LMS and Scalameta are incompatible in terms of the Scala compiler versions.

Threats to validity There are two major threats to the validity of our evaluation. The first threat is that measuring conciseness by counting SLOC may not be fair especially when different languages are used. We mitigate this threat by making the code style and the maximum character-per-line consistent for each implementation. The second threat is the representativeness of the TAPL interpreters. They are small languages for teaching purposes. It might still be questionable whether Castor scale to model larger languages that are actually used in practice. Nevertheless, TAPL interpreters have already covered a lot of core features that are available in mainstream languages.

4.8 Case Study II: UML Activity Diagrams

In Section 4.7, we have evaluated the functional aspects of Castor. In this section, we evaluate the imperative aspects of Castor. To do so, we conduct another case study on a subset of the UML activity diagrams, which can be seen as a richer language than the FSM language discussed in Section 4.5. This case study examines hierarchical datatypes, imperative visitors and graphs.

4.8.1 Overview

An execution model of UML activity diagrams has been proposed as one of the challenges of the Transformation Tool Contest (TTC'15).

Metamodel Figure 4.10 shows the metamodel of UML activity diagrams, where ***Name*** denotes abstract classes and **Name** denotes concrete classes. An `Activity` object represents an instance of a UML activity diagram, which contains a sequence of `ActivityNodes` and `ActivityEdges`. `ExecutableNode` and `ControlNode` are two intermediate types of `ActivityNode` for classifying nodes that perform actions or control the flow. There are several concrete nodes. `InitialNode` and `ActivityFinalNode` are the start/end of activity diagrams; `DecisionNode` and `MergeNode` are the start/end of alternative branches; `ForkNode` and `JoinNode` are the start/end of concurrent branches. On the other hand, `OpaqueAction` sequentially executes a sequence of `Expressions`. `ActivityNodes` are connected by `ActivityEdges`. Similar to `GuardedTrans` discussed in Section 4.5.3, a `ControlFlow` is a specialized `ActivityEdge`, which is guarded by the current `BooleanValue` stored in a `BooleanVariable`. `Expressions` are also organized in a hierarchical way according to their types (`Boolean` or `Integer`) and the number of operands (`Unary` or `Binary`).

Goal and challenges The goal is to extend this simplified metamodel of UML activity diagrams with the dynamic execution semantics. The semantics is defined by performing transitions on activity nodes step by step using an imperative style. Several *runtime concepts* need to be introduced. Adding these runtime concepts poses two modularity challenges: *operation extensions* and *field extensions*. One example of an operation extension is `execute`, which is added to the `Expression` hierarchy for executing the calculation. One example of a field extension is a mutable boolean value `running`, which is added to `ActivityNode` for distinguishing triggered nodes from others.

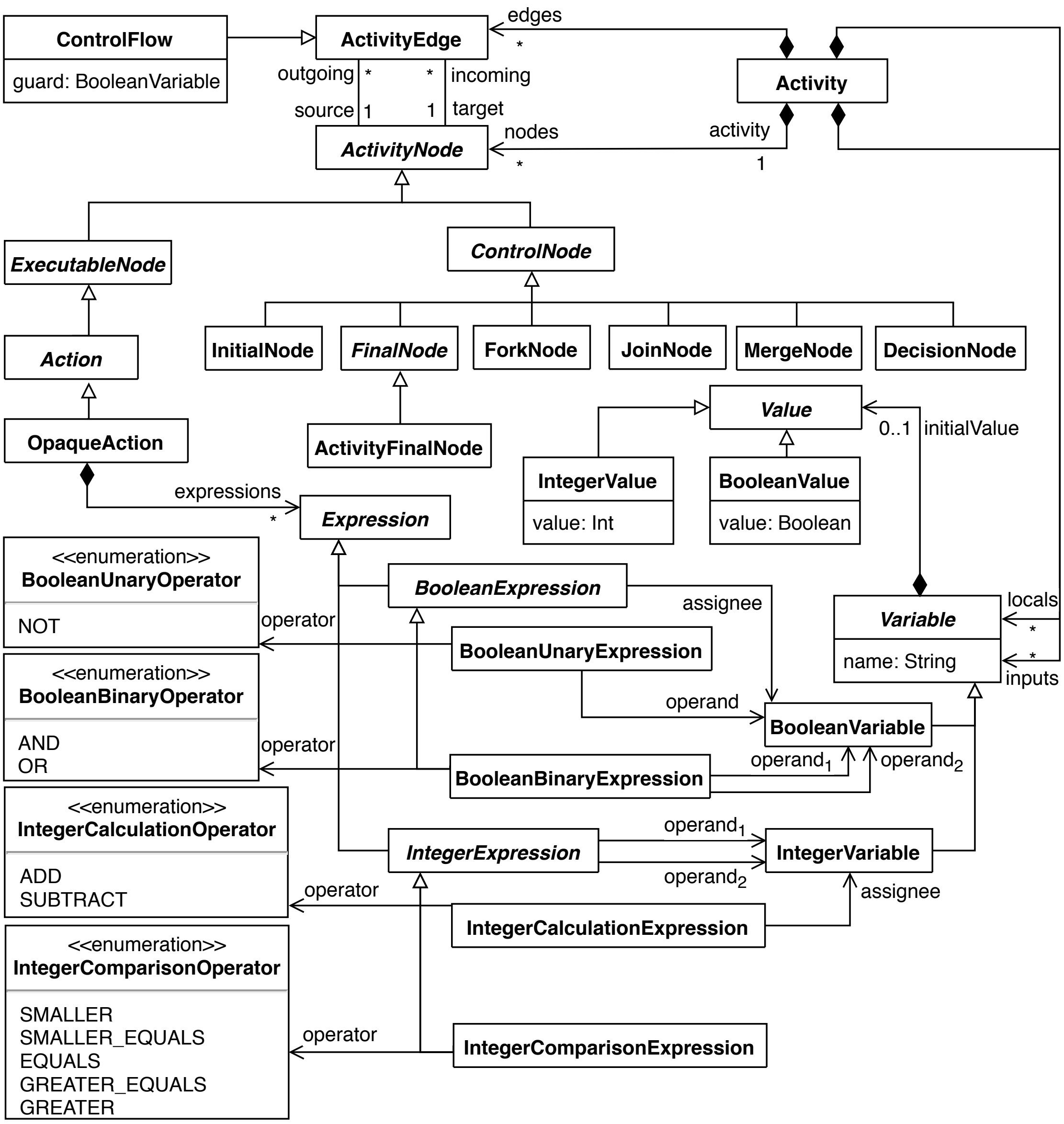


Figure 4.10: *Metamodel of UML Activity Diagrams, an excerpt adapted after the TTC'15 document [Mayerhofer and Wimmer, 2015].*

Reference implementation The reference implementation[5] is written in Java with EMF [Steinberg et al., 2008]. The metamodel is described in Ecore from which Java interfaces are generated. Then semantics is encoded by defining classes that implement those interfaces using the Interpreter pattern [Gamma et al., 1994]. The reference is non-modular because the Interpreter pattern facilitates adding new classes but lacks the ability to add new operations. Therefore, the reference implementation has to anticipate the operations on the metamodel. Moreover, consistent with what Figure 4.10 shows, operators were modeled as enumerations and recognized using `switch-case` clauses in Java, which are closed for extensions.

Refactored implementation Our refactoring only focuses on the metamodel and the semantics parts. Since the original implementation is written in Java, we first port it into Scala and then refactor it using Castor. Figure 4.11 gives an overview of the refactored implementation, which consists of four Castor components. Concretely, we make the following changes to the ported implementation for increasing modularity:

1. **Separate metamodel and operations.** With Castor, we do not need to foresee the operations on the metamodel since operations can be modularly added afterwards. Thus, the refactored implementation separates the metamodel and operations upon it respectively in ***Model** and ***Lang**.

2. **Expression language as an independently reusable component.** Values, variables and expressions are essentially a sublanguage independent of the UML activity diagrams. Instead of defining the expression sublanguage together with UML activity diagrams within a single `@family` component, we extract its metamodel into **ExpModel** and its semantics into **ExpLang** and let **UmlModel** and **UmlLang** extend them respectively. This allows the expression sublanguage to be reused or extended individually.

3. **Overridden methods as visitors.** Methods that are overridden in the subclasses are rewritten as visitors, such as `isReady` and `fire` on `ActivityNode` and `execute` on `Expression`. Since only a few cases of `isReady` and `fire` are overridden whereas every case of `execute` is overridden, we use the default visitor (annotated as `@default`) for the former and the ordinary visitor (annotated as `@visit`) for the latter. For non-overridden methods, we move them out of a class and use an explicit argument to capture `this`.

4. **Operators as open datatypes.** Operators are refactored as `@adt` hierarchies and their semantics are given by visitors for enabling extensions. This allows new kinds of operators such as multiplication to be added later.

4.8.2 Evaluation

We evaluate Castor by answering the following questions:

[5] https://github.com/moliz/moliz.ttc2015

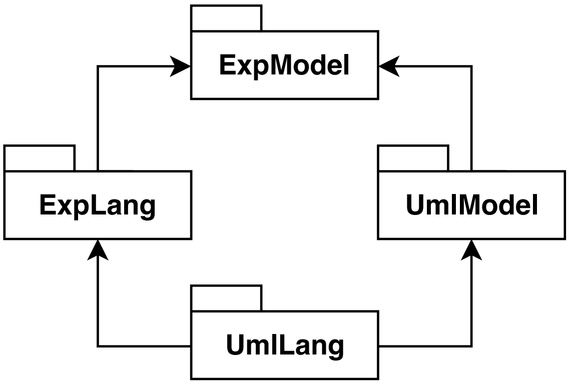


Figure 4.11: *Refactored implementation.*

- **Q1.** Does the refactoring preserve the behavior of the ported implementation?
- **Q2.** Can CASTOR solve the modularity challenges?
- **Q3.** How does the refactoring affect the SLOC?
- **Q4.** Is the performance overhead reasonable?

Q1 To make sure that our refactoring does not affect the correctness of the implementation, we ran the test suite provided by the TTC'15 document. The test suite contains 6 small activity diagrams where all kinds of `ActivityNodes` and `Expressions` are covered. The refactored implementation passes all the tests in the test suite. This gives us some confidence that the refactored implementation preserves the behavior of the ported implementation.

Q2 For the operation extension challenge, the answer is yes. Operations are added by defining new visitors, which are fully modular. However, CASTOR does not address the field extension challenge very well. With the current version of CASTOR, we cannot extend existing classes with additional fields while keeping their names. The workaround is to introduce subclasses of different names. For example, if we want to extend `ActivityNode` with a field called `running`, we have to define a new class called `RuntimeActivityNode` that extends `ActivityNode` with `running`. The drawback is that `RuntimeActivityNode` and `ActivityNode` coexist and all existing operations need to be modified for handling `RuntimeActivityNode`. It is possible to have an alternative design for CASTOR, which does not introduce a new name while accomplishing field extensions in CASTOR. However, this brings some other complications. Such alternative design is discussed in Section 4.9.

Q3 The SLOC of the ported version and the refactored version are 489 and 411 respectively. Surprisingly, the refactoring brings extra modularity while reducing the SLOC. One reason is that in the ported version, methods are first declared in traits and then implemented in classes while the refactored version needs no prior declarations. Another reason is that by properly using CASTOR's default visitors and combined visitors, some def-

Table 4.3: *Performance evaluation in milliseconds.*

Name	Description	INTERPRETER	CASTOR
test_1	1000 sequential actions	22.1	56.6
test_2	100 parallel branches each with 10 actions	20.7	39.8
test_3	Similar to test_2 with a variable increased	22.8	39.9

initions can be shortened. For example, `Execute` in the refactored version is a combined visitor for `Expression` and 4 operators.

Q4 We reuse the test suite provided by the TTC'15 document, which includes 3 large activity diagrams for measuring the performance. Table 4.3 gives a simple description for each test case and the average execution time for 10 runs (measured in milliseconds) for the two implementations. The benchmark is executed using the same machine specified in Section 4.7. The CASTOR's implementation is around 2 to 3 times slower than the non-modular ported implementation. These results are similar to the results we get in Section 4.7 and further confirm that CASTOR's modular implementation introduces an acceptable performance penalty.

Threats to validity One threat to the validity of the evaluation is that the test suite is very small and might not be able to find out bugs that are introduced by refactoring. Also, directly comparing a CASTOR's implementation with respect to the reference implementation may be unfair since different programming languages are used. To exclude such language-wise factor on evaluation, we compared to the ported Scala implementation. As our focus is on the semantics part, irrelevant code like parsing is ignored.

4.9 Design Options

In this section, we briefly discuss other design options and their compromises.

Nested patterns There is an alternative way of writing nested patterns. For example, `tmIf` can be rewritten in the following way:

```
override def tmIf = x => x.t1 match {
  case TmTrue => x.t2
  case TmFalse => x.t3
  case t1 => TmIf(this(t1),x.t2,x.t3)
}
```

Instead of directly pattern matching on an `TmIf` object, we capture it first using a variable `x` and then explicitly `match` on its subterm `t1`. For the case of `tmIf`, this alternative implementation is arguably less intuitive than the version we presented in Section 5.2. Nevertheless, this approach comes in handy when: 1) the object being matched contains a lot of fields and most of them are not interesting in nested patterns; 2) there are a lot of `case` clauses for nested patterns and repeating the top-level pattern in each `case` clause becomes tedious.

Specialized visitors Programming with visitors can be simplified using specialized visitors. The default visitors generated by Castor (annotated as `@default`) are an instance. In fact, there are more such specialized visitors. For example, visitors can be combined with *visitor combinators* [Visser, 2001]; boilerplate for querying and transforming data structures can be eliminated by *traversal templates* [Zhang and Oliveira, 2017]. Essentially, these specialized visitors can also be generated by Castor. Currently, only default visitors are generated because 1) in our experience they are most frequently used; 2) generating all other infrequently used specialized visitors increases the time of code generation and the size of generated code. Ideally, specialized visitors should be generated by need. Limited by current Scalameta, this is impossible for the moment.

Refinable variants As our visitor encoding shows, the key to extensibility is capturing concrete types with bounded type members for allowing future refinements. The same idea can also be applied to variants, where the visitor method signature refers to a type member instead of a class name. By doing this, we are able to extend that class with additional fields seamlessly by covariantly refining the type member to the new class. An application of refinable variants would be guarded transitions discussed in Section 4.5.3:

```
class Trans(event: String, to: State, var tm: Tm[Boolean] = TmTrue)
  extends super.Trans(event, to)
```

Instead of adding a new variant called `GuardedTrans`, we refine the existing `Trans`. The benefit is that existing visitors that do not concern about the additional parameter `tm` can be unchanged. In contrast, for the case of `GuardedTrans`, we have to update all existing visitors with an implementation of `guardedTrans`. However, the downside of supporting refinable variants in Castor is that it brings more book-keeping burden on variants for the user. We consider the price to pay is higher than the benefit it brings.

4.10 Conclusion

In this chapter, we have presented Castor, a Scala framework for programming with extensible, generative visitors using simple annotations. Visitors written with Castor are type-safe, concise, exhaustive, extensible and composable. Moreover, both functional and imperative style visitors are supported. We have shown how to use Castor in designing a better pattern matching mechanism in an OO context, developing modular well-typed EDSLs, doing extensible programming on graphs, etc. The effectiveness of Castor is validated by our case studies on TAPL interpreters and UML activity diagrams.

Chapter 5

Compositional Programming

In this chapter, we shift our focus on novel language design for modularity. We propose a new statically typed modular programming style called *Compositional Programming*. Compositional Programming offers an alternative way to model data structures that differs from both algebraic datatypes in functional programming and conventional OOP class hierarchies. We introduce four key concepts for Compositional Programming: *compositional interfaces*, *compositional traits*, *method patterns* and *nested trait composition*. Altogether these concepts allow us to naturally solve challenges such as the EP, model attribute-grammar-like programs and generally deal with modular programs with complex dependencies. We present a language design called CP, which is proved to be type-safe, together with several examples and three case studies.

5.1 Introduction

An important aspect of programming is how to define data structures and the operations over those data structures. Different language designs offer different mechanisms for this purpose. OOP languages model data structures using class hierarchies and techniques such as the Composite pattern [Gamma et al., 1994]. Typically, there is an interface that specifies all the operations (methods) of interest for the data structure. Multiple classes implement different types of nodes in the data structure, supporting all the operations in the interface. Many functional languages employ *algebraic datatypes* [Burstall et al., 1981] to model data structures, and use functions (typically defined by pattern matching) to model the operations over those data structures. As widely acknowledged by the EP [Wadler, 1998], both algebraic datatypes and class hierarchies have modularity problems. With algebraic datatypes and pattern matching, adding new operations is easy, but adding constructors is hard.

This chapter presents a new statically typed modular programming style called *Compositional Programming*. In Compositional Programming, there is no EP: it is easy to get extensibility in two dimensions (i.e. it is both easy to add new constructors, as well as new operations). Compositional Programming offers an alternative way to model data structures that differs from both algebraic datatypes in functional programming and conventional OOP class hierarchies. The key ideas of Compositional Programming are implemented in a new programming language called CP. For example, the code for modeling arithmetic expressions can be expressed in CP as:

```
type ExpSig<Exp> = {        -- Compositional interface for expressions with a sort Exp
  Lit : Int -> Exp;         -- Constructor (returns the sort Exp)
  Add : Exp -> Exp -> Exp;  -- Constructor (returns the sort Exp)
};

type Eval = { eval : Int };         -- Concrete type for evaluation (instantiates Exp)
evalNum = trait implements ExpSig<Eval> => {     -- Compositional trait for evaluation
  (Lit     n).eval = n;                          -- Definition using a method pattern
  (Add e1 e2).eval = e1.eval + e2.eval;          -- Definition using a method pattern
};
```

In the CP code above, the definition ExpSig<Exp> (a compositional interface) plays a similar role to the algebraic datatype in functional languages. The special type parameter Exp in ExpSig<Exp> is called a *sort*, and models types that represent datatypes in CP. All constructors in CP must have a return type which is a sort. Like in functional languages, in CP adding new operations is easy using a special form of *traits* [Schärli et al., 2003]. Unlike functional programming, where adding new constructors is difficult and non-modular, in CP the addition of new constructors is also easy. We will later illustrate the modularity of CP in detail. In essence, there are four new key concepts in CP:

- **Compositional interfaces** can be viewed as an extension of traditional OOP interfaces, such as those found in languages like Java or Scala. In addition to declaring method signatures, compositional interfaces also allow the specification of *constructor signatures*. In turn, this enables *programming the construction of objects against an interface*, instead of a concrete implementation. Compositional interfaces can be parametrized by sorts, which abstract over concrete types, and are used to determine which kind of object is built.

- **Compositional traits** extend traditional *traits* [Schärli et al., 2003] and *first-class traits* [Bi and Oliveira, 2018]. Compositional traits allow not only the definition of (virtual) methods but also the definition of *virtual constructors*. They also allow nested traits, which are used to support *trait families*. Trait families are akin to class families in *family polymorphism* [Ernst, 2001].

- **Method patterns**, such as (Lit n).eval, provide a lightweight syntax to define method implementations for nested traits, which arise from virtual constructors. This enables compact method definitions for trait families, which resemble programs defined by pattern matching in functional languages, and programs written with attributes in *attribute grammars* [Knuth, 1968, 1990].

- **Nested trait composition** is the mechanism used to compose compositional traits. The foundations for this mechanism, originate from nested composition, which has been investigated in recent calculi with disjoint intersection types and polymorphism [Oliveira et al., 2016; Alpuim et al., 2017; Bi et al., 2018, 2019]. We show how nested composition can smoothly be integrated into compositional traits. Nested trait composition plays a similar role to traditional class inheritance, but generalizes to the composition of nested traits. Thus it enables a form of inheritance of whole hierarchies, similar to the forms of composition found in family polymorphism. Nested trait composition is *associative* and *commutative* [Bi et al., 2018], just like the composition for traditional traits [Schärli et al., 2003].

Altogether, these concepts allow us to solve various challenges naturally. For instance, they enable a very natural and simple solution to the EP. More interestingly, Compositional Programming can deal with harder modularity challenges, such as modeling interesting classes of attribute grammars, which often contain non-trivial dependencies. Such attribute-grammar-like programs can be expressed in a statically safe way and without giving up the modularity benefits of attribute grammars. More generally, Compositional Programming offers a range of mechanisms to deal with modular programs with various *complex forms of dependencies*.

The CP language is inspired by the SEDEL language [Bi and Oliveira, 2018]. The main novelties over SEDEL are the four compositional programming mechanisms listed above: compositional interfaces, compositional traits, method patterns, and nested trait composition. Compositional Programming is partly inspired by *generalized Object Algebras* [Oliveira et al., 2013], but the built-in language mechanisms make modular programming natural, fully statically typed, and without boilerplate code and excessive parametrization. We present several examples and three case studies in CP. We also introduce a technique called *polymorphic contexts* to deal with components that require some form of context in a modular way. In turn, polymorphic contexts are helpful to model L-attributed grammars. Our first case study is on the design of an Embedded Domain-Specific Language (EDSL) for circuits [Hinze, 2004; Gibbons and Wu, 2014]. This EDSL is interesting because it has various extensions that can be modularly defined, as well as various dependencies between components. Our second case study is a mini interpreter, which is a larger study and can be extended in several ways. The last case study is an implementation of the C0 compiler, inspired by the work of Rendel et al. [2014]. In this case study, various extensions can be formulated as attributes and those attributes contain non-trivial dependencies to other attributes.

Finally, we present a small calculus that captures the essence of CP. This calculus is shown to be type-safe via an elaboration to the F_i^+ calculus [Bi et al., 2019], which is a recently proposed calculus that supports *disjoint intersection types* [Oliveira et al., 2016], *disjoint polymorphism* [Alpuim et al., 2017] and *nested composition* [Bi et al., 2018].

In summary, the contributions of this chapter are:

- **Compositional Programming:** We propose a new programming style that encourages weaker dependencies and increases the modularity of programs. Compositional Programming eliminates the EP and can deal with modular programs with complex dependencies.

- **A language design for Compositional Programming:** We present a concrete language design in the form of the CP calculus. The semantics of this calculus is given by elaboration to the F_i^+ calculus and we prove the *type safety* of the elaboration.

- **Attribute Grammars in CP:** We show that CP is powerful enough to implement programs with attributes that are expressible in attribute grammars [Knuth, 1968, 1990] in a concise and fully statically type-checked manner. Our technique is partly inspired by the encoding of Rendel et al. [2014], but CP avoids explicit definitions of composition operators, which are necessary in Rendel et al. [2014]'s encoding.
- **Polymorphic contexts:** We introduce a simple technique that combines disjoint polymorphism and Compositional Programming to allow for modular contexts in modular components.
- **Implementation, case studies, and examples:** We have an implementation of CP. We present several examples and three case studies in CP. Altogether, these examples and case studies illustrate how to naturally solve challenges such as the EP or modeling attribute-grammar-like programs. The implementation and case studies can be found in:

https://github.com/wxzh/CP

5.2 An Overview of Compositional Programming

This section presents an overview of Compositional Programming. We start by introducing the basic mechanisms of Compositional Programming with the EP. We then move on to harder modularity issues, such as various forms of dependencies, which arise in modular programs that compute various forms of attributes. All the CP programs presented here can run in our implementation of CP.

5.2.1 The Expression Problem with Compositional Programming

In Compositional Programming, there is no EP: it is both easy to add new variants, as well as new operations. Indeed, an explicit goal of Compositional Programming is that programmers do not need to face the tension of choosing one dimension for extensibility. Therefore, Compositional Programming offers an alternative way to model data structures that differs from both algebraic datatypes and conventional OOP class hierarchies. Because of such fundamental differences, and the more modular programming style, we can think of Compositional Programming as an alternative programming paradigm We introduce the mechanisms used in the programming language CP by solving the EP next.

Compositional interfaces and sorts In CP, we use a compositional interface to declare a datatype. Compositional interfaces generalize conventional OOP interfaces: we can define not only the signatures of methods but also those of constructors, which is not possible in conventional OOP languages like Java. The compositional interface for basic arithmetic expressions is:

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
```

There are two kinds of expressions for now: literals and addition. The type parameter `Exp` wrapped in angle brackets is called a *sort* of the compositional interface, working as the type of both kinds of expressions. `Lit` and `Add` are the signatures of the *constructors* for arithmetic expressions. In CP, constructors always start with a capital letter, while all methods start with a lowercase letter. The return type of a constructor must always be a sort, and there can be arbitrarily many sorts in a compositional interface.

Note that sorts in compositional interfaces are handled differently from normal type parameters, which is why they have a special syntax of angle brackets around them. In essence, compositional interfaces are based on several ideas related to generalized Object Algebras (see 2.4.1), and the special treatment for sorts involves, among other things, distinguishing uses of positive and negative occurrences of the type variables. The semantics of sorts is discussed in detail in Section 5.4. Nonetheless, these subtle semantic differences are essentially only relevant for programs with dependencies, such as those presented in Section 5.2.2. For now, it suffices to think of sorts as type parameters.

Analogy with algebraic datatypes Compositional interfaces defining only constructors play a similar role to algebraic datatypes in functional programming. Like algebraic datatypes, they can be used to define data structures by specifying the name of the data structure and the signatures of the constructors. For example, a Haskell counterpart of the compositional interface above is:

```
data Exp where
  Lit :: Int -> Exp
  Add :: Exp -> Exp -> Exp
```

The sort `Exp` of the compositional interface corresponds to the name of the datatype, whereas in both cases the constructors essentially specify the same information: the signatures of the constructors.

Compositional traits The unit of code reuse in CP is a generalization of *(first-class) traits* [Schärli et al., 2003; Bi and Oliveira, 2018]. The generalization stems from the fact that we can define not only method implementations but also *(virtual) trait constructors* within a trait. In CP, the implementation of the `eval` operation for the basic form of expressions can be done in a trait:

```
type Eval = { eval : Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
```

This trait implements the compositional interface `ExpSig` with the sort instantiated as `Eval`, corresponding to its actual operation. `Eval` is another example of a compositional interface, which in this case, because it has no sorts, just degenerates into a conventional OOP-style interface with a method `eval` returning `Int`. Within the trait `evalNum`, we implement the `eval` method for `Lit` and `Add` to evaluate the arithmetic expressions.

Method patterns The *method pattern* `(Lit n).eval` is a lightweight syntax used to define the `eval` method within the trait (which implements the interface `Eval`) that the constructor `Lit` returns. In CP, method patterns are used to make trait definitions concise. Method patterns allow definitions that resembles functional programming definitions by pattern matching, or attributes in attribute grammars. An alternative way of defining constructors would explicitly use first-class traits, which is essentially what method patterns are desugared to:

```
evalNum = trait implements ExpSig<Eval> => {
  Lit     n = trait => { eval = n; };
  Add e1 e2 = trait => { eval = e1.eval + e2.eval; };
};
```

Nested traits and trait families As shown above, method patterns inside a trait are essentially defining *nested traits*. The outer trait, `evalNum`, is called a *trait family*. The terminology *trait family* is borrowed from *family polymorphism* [Ernst, 2001]. In family polymorphism, a family is a class that contains nested virtual classes. In CP, a trait family is a trait that contains nested virtual traits.

Virtual constructors One significant difference from most existing languages is that CP's constructors are *virtual*. In languages like Java, there are virtual methods, whose concrete implementation is unknown at the time a class is defined. In this way, the references to methods are loose and determined only when objects are instantiated. However, languages like Java do not support virtual constructors or *virtual classes* [Madsen and Moller-Pedersen, 1989]. A *virtual constructor* is not bound to a specific implementation of a trait. Instead, it conforms to the signature in the compositional interface. With virtual constructors, it is possible to create a trait that constructs an expression *without sticking to a particular implementation* of the compositional interface:

```
expAdd Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Lit 4) (new Lit 8);
};
```

In CP, term parameters start with a lowercase letter while type parameters are capitalized, so `Exp` is a type parameter, serving as the sort of `ExpSig`. Thus `expAdd` contains an abstract expression that is not associated with any concrete method implementations.

Self-type annotations The `expAdd` trait above is parameterized by `Exp` and specifies its self-type as `ExpSig<Exp>`. Such self-type annotations are similar to those in Scala [Odersky et al., 2004], which enable a modular way of injecting dependencies on other operations/-constructors. The self-type annotation of `expAdd` expresses that `expAdd` must be merged with some trait that concretely implements `ExpSig` for instantiation. By specifying the self-type, the constructors declared inside `ExpSig`, i.e. `Lit` and `Add`, are directly available for building an expression named `test`. Section 5.2.2 will discuss additional mechanisms in CP to deal with other forms of dependencies.

Extensibility: adding new operations Like in functional programming with algebraic datatypes and pattern matching, adding a new operation is trivial in CP. We can just create an independent trait family that implements the compositional interface:

```
type Print = { print : String };
printNum = trait implements ExpSig<Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

The trait `printNum` implements `ExpSig<Print>` and defines `Lit` and `Add` using method patterns, in a way similar to `evalNum`. It modularly supports pretty-printing for expressions.

Nested trait composition At the heart of Compositional Programming is a powerful composition mechanism called *nested composition* [Bi et al., 2018]. Nested composition allows the composition of multiple trait families. This mechanism can be viewed as a form of multiple (trait) inheritance, but with the added ability to recursively compose nested traits automatically. Thus it provides composition mechanisms that are similar to the ones found in languages with family polymorphism. Like trait composition [Schärli et al., 2003], nested composition in CP is *associative* and *commutative* [Bi et al., 2018]. Furthermore, following the trait model and SEDEL, conflicts arising during composition are rejected. In CP, which is statically typed, such conflicts are statically rejected as type errors, following a similar approach to trait composition in SEDEL. Conflict resolution in CP can be done using method overriding or an explicit exclusion operator (like in many models of traits).

In CP, nested composition is performed by the merge operator `,,` [Dunfield, 2014]. For example, to compose the trait families `evalNum` and `printNum` with `expAdd` we can write:

```
e = new evalNum ,, printNum ,, expAdd @(Eval&Print);
```

Note that the merge operator `,,` binds tighter than `new` (but application and type application still bind tighter than `,,`). The type application `expAdd @(Eval&Print)` makes the generic trait `expAdd` concrete by instantiating the type parameter `Exp` as the argument `Eval&Print`. Since the self-type annotation of `expAdd` has been instantiated as `ExpSig<Eval&Print>`, in order to meet this self-type requirement, the trait has to be merged with some trait that simultaneously implements the `eval` and `print` operations for the constructors exposed by `ExpSig`. The requirement is met by merging `expAdd @(Eval&Print)` with `evalNum` and `printNum`. Thus, the merged trait is successfully instantiated into an object using a `new` expression.

It is useful to take a moment and understand why the trait composition performed for `e` and the method calls in the expression above pass type-checking and work as expected. Note that the types of `evalNum`, `printNum` and `expAdd @(Eval&Print)` are, respectively, `Trait[ExpSig<Eval>]`, `Trait[ExpSig<Print>]` and `Trait[ExpSig<Eval&Print>,{test:Eval&Print}]`. Their merge is then of the type `Trait[ExpSig<Eval&Print>,ExpSig<Eval>&ExpSig<Print>&{test:Eval&Print}]`. The merged trait type is an *instantiatable* trait type (i.e. the provided type is a subtype of the required type) because `ExpSig<Eval>&ExpSig<Print>` is a subtype of `ExpSig<Eval&Print>` in CP.

In essence, the subtyping relation employed by CP supports *distributivity* of intersections over other constructs [Bi et al., 2018]. In CP, intersections can distribute over function types, records, and trait types. Through the object `e`, we can both evaluate and print the addition expression:

```
e.test.print ++ " is " ++ e.test.eval.toString  --> "(4+8) is 12"
```

Extensibility: adding new variants Finally, we show how to add multiplications to the language modularly. Note that this is where Compositional Programming differs from algebraic datatypes and definitions by pattern matching, where such extensions cannot be modularly added. We first define a compositional interface that extends `ExpSig` with a `Mul` constructor:

```
type MulSig<Exp> extends ExpSig<Exp> = {
  Mul : Exp -> Exp -> Exp;
};
```

We then implement evaluation and printing by defining two trait families `evalMul` and `printMul` that respectively inherit `evalNum` and `printNum` and complement a definition for `Mul`:

```
evalMul = trait implements MulSig<Eval> inherits evalNum => {
  (Mul e1 e2).eval = e1.eval * e2.eval;
};
printMul = trait implements MulSig<Print> inherits printNum => {
  (Mul e1 e2).print = "(" ++ e1.print ++ "*" ++ e2.print ++ ")";
};
```

Without editing any existing code, we modularly add new data variants. Note that here we use a new keyword **`inherits`**. In CP, inheritance is based on the merge operator but given a more convenient syntax that resembles conventional OOP. Besides nested composition, **`inherits`** additionally allows fields defined in the parent trait to be overridden but still can be used via **`super`** calls in the trait body. With **`super`** calls and inheritance, we can, for instance, construct a slightly more complex expression:

```
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp => {
  override test = new Mul super.test (new Lit 4);
};
```

The trait `expMul` inherits `expAdd`, refines the self-type to `MulSig` and overrides the `test` field. The overridden expression reuses the inherited expression via **`super`**`.test`. Finally, the object `e'` supports all three constructors together with the evaluation and printing operations.

Summary For the basic EP, there are already many solutions in the literature, including some in mainstream programming languages [Garrigue, 2000; Torgersen, 2004; Oliveira et al., 2006a; Zenger and Odersky, 2005; Swierstra, 2008; Oliveira and Cook, 2012]. One interesting aspect of the solution presented here is that it is quite elegant and natural, while solutions in mainstream languages tend to have significant amounts of boilerplate code, or are written in highly parametrized code that is not programmer-friendly. Additionally, the more interesting aspect of Compositional Programming is its wide support for various forms of modular code with dependencies, which is shown next.

5.2.2 Dependencies and S-Attributed Grammars

In the original EP by Wadler [1998] there are very few *dependencies*. In particular, the operations (`eval` and `print`) depend only on themselves, but they do not depend on other operations. Matters become significantly more complicated in the presence of more advanced forms of *dependencies*, and very few existing solutions to the EP have effective mechanisms to deal with dependencies in a modular way. Two primary mechanisms are used to deal with dependencies in CP:

- **Compositional interface type refinement:** When a trait implements some compositional interface, it can refine the *sort types in input positions*. This allows the child nodes in a data structure to assume some functionality that is not implemented in the enclosing trait, but will eventually be part of the final composition later.
- **Self-type annotations:** Like in Scala, the types of self-references can be specified/refined. This enables the self-references to assume functionality that will be implemented by a different trait that will eventually be composed with the current trait. In the context of trait families, there are two kinds of *self-references*: *object self-references* and *family self-references* [Oliveira et al., 2013]. The trait `expAdd` in Section 5.2.1 already illustrates family self-references, so in what follows we illustrate only object self-references.

We use examples inspired from *S-attributed grammars* [Knuth, 1968] and the work from Rendel et al. [2014] to illustrate how Compositional Programming addresses programs with different kinds of dependencies in a modular way. S-attributed grammars deal with synthesized attributes, which are computed from the children. Compositional Programming also allows dependencies on self. The simplest form of dependencies is dependencies on the same attribute of children, which occurred several times in Section 5.2.1. Therefore, we focus on two kinds of non-trivial dependencies on other synthesized attributes, which hereinafter we call *child dependencies* and *self dependencies*.

Child dependencies Child dependencies occur when an attribute depends on other synthesized attributes of the children. Here is the `printChild` example from Section 4.1 in CP:

```
printChild = trait implements ExpSig<Eval % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

Here (Add e1 e2).print depends on e2.eval. However, there is no implementation of eval in the trait printChild. To make this child dependency feasible, we use compositional interface type refinement: we change the concrete interface being implemented to ExpSig<Eval % Print> (instead of just using ExpSig<Print>). This syntax means that the input type for the expressions in the Add constructor (and other constructors, if any, referring to the sort Exp) is Eval&Print, while the output type is still Print. By doing this, we enable the child nodes of Exp to depend on an attribute whose implementation is modularly defined somewhere else. Two examples of trait instantiations are:

```
new printChild ,, expAdd @Print                    -- Type Error!
new printChild ,, evalNum ,, expAdd @(Print&Eval)  -- OK!
```

This first instantiation attempt fails type-checking because it depends on eval, which is missing. The second one works since we merge printChild with the trait evalNum (which implements Eval). Importantly, instead of evalNum, we could have used any implementation with the same type.

Contrast with inheritance It is useful to compare the previous example with the more common OOP approach based on inheritance:

```
printInh = trait implements ExpSig<Eval&Print> inherits evalNum => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

The first thing to notice is that the code in the trait body of printIhn is exactly the same as that in printChild. Conforming to the compositional interface instantiated with Eval&Print, printInh essentially implements both eval and print instead of only print, because the evalNum trait family is inherited. Although both approaches work, printInh is tightly coupled with the particular implementation of evaluation coming from evalNum. In contrast, printChild declares a weaker dependency on the abstract interface of Eval. We can delay the combination with a concrete implementation of Eval until the instantiation phase. In short, printChild allows for *weak* child dependencies, which are not coupled with a particular implementation, while preserving strong static type safety. More generally, most uses of inheritance in CP can be converted into code that has weaker dependencies.

Self dependencies A second interesting case is dependencies on other synthesized attributes of the self-reference. In the following example, the attribute print depends on self.eval:

```
printSelf = trait implements ExpSig<Eval % Print> => {
  (Lit     n            ).print = n.toString;
  (Add e1 e2 [self:Eval]).print = if self.eval == 0 then "0"
                                  else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

To deal with this dependency without sticking to a particular implementation of eval, we add an (object) self-type annotation [self:Eval] to use self.eval in Add. Note that we also need to change the sort instantiation to `ExpSig<Eval % Print>` like the child dependencies, in order to change the self-type of the returning trait correspondingly. The static type-checker will check whether the trait is later merged with another trait that implements `ExpSig<Eval>`. With no compromises on type safety, CP enables modular weak self dependencies on other attributes.

Mutual dependencies Finally, a more general form of dependencies is *mutual dependencies*, which happen when two attributes are inter-defined, i.e., they depend on each other. Mutual dependencies can involve both child and self dependencies, as illustrated in the following example:

```
type PrintAux = { printAux : String };
printMutual = trait implements ExpSig<PrintAux % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = e1.printAux ++ "+" ++ e2.printAux;
};
printAux = trait implements ExpSig<Print % PrintAux> => {
  (Lit     n [self:Print]).printAux = self.print;
  (Add e1 e2 [self:Print]).printAux = "(" ++ self.print ++ ")";
};
```

The two trait families `printMutual` and `printAux` cooperate to omit the outermost parentheses in pretty-printing. We can see that `(Add e1 e2).print` depends on the `printAux` while `printAux` depends on `print`, thus `print` and `printAux` are mutually dependent. CP handles such mutual dependencies modularly. We can combine the traits and use them as before:

```
(new printMutual ,, printAux ,, expAdd @(Print&PrintAux)).test.print  --> "4+8"
```

Summary Many attribute grammar systems allow the modular definition of attributes, but this is usually done at the cost of modular type-safety. The compositional mechanisms in CP retain the ability of modularly defining attributes from S-attributed grammars or even attributes from self-references, but in a statically type-safe setting. Moreover, the implementations of the attributes are changeable in the final composition. For example, the aforementioned traits `printNum`, `printChild`, `printSelf`, and `printMutual` all implement the `print` method. A programmer can freely pick his favorite implementation to combine with other attributes, such as `eval`.

5.3 Parametric Polymorphism and L-Attributed Grammars

In Section 5.2, we introduced the basic mechanisms for Compositional Programming and illustrated how CP deals with dependencies. One feature that practically all modern languages support is some form of *parametric polymorphism* (or *generics* in OOP languages). A reasonable question to ask is how Compositional Programming interacts with parametric polymorphism. Furthermore, one may wonder if the combination of parametric polymorphism and Compositional Programming enables novel techniques that are useful for programming.

In this section, we explore this question. CP has full support for a form of parametric polymorphism called *disjoint polymorphism* [Alpuim et al., 2017]. We introduce a novel technique called *polymorphic contexts*, which addresses the problem of different modular components requiring different kinds of contextual information. The technique provides encapsulation of contexts: modular components have access to the contextual information they require but do not have access to contextual information used by other components. This technique is useful to model *inherited attributes*, where it is possible to access attributes from parents or siblings. To achieve this, a context should be attached to pass attributes from top to bottom. There is a close relationship between contextual evaluation and *L-attributed grammars* [Knuth, 1968; Rendel et al., 2014].

5.3.1 Contexts and Modular Components

There are plenty of scenarios where we need to add contexts to our code. One of the most common examples is variable binding in an interpreter. Recall that, in Section 5.2.1, we defined the type of `eval` as `Int`. But this interface is not suitable for an expression language with variable binding. A naive fix is to modify the existing code and add a context to all the trait families:

```
type Eval = { eval : EnvN -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval (env:EnvN) = n;
  (Add e1 e2).eval (env:EnvN) = e1.eval env + e2.eval env;
};
evalVar = trait implements VarSig<Eval> => {
  (Let s e1 e2).eval (env:EnvN) = e2.eval (insert @Int s (e1.eval env) env);
  (Var       s).eval (env:EnvN) = lookup @Int s env;
};
```

Note that we add an `EnvN` parameter to `eval`. Since CP's support for type inference is still limited, while `n`, `e1` and `e2` do not require type annotations, we have to annotate `env` with `EnvN` here. `EnvN` is a map from `String` to `Int`, serving as the variable environment, while `insert` and `lookup` are auxiliary functions on maps. For the `Let` expression in `evalVar`, we evaluate `e1` and insert the evaluation result into `env` before evaluating `e2` (the body of the `Let` expression). For the `Var` expression, we look up the variable name to get its value. Although `evalNum` does not need the context for evaluating arithmetic expressions, we still have to pass the context to the recursive calls to make `env` available everywhere.

So far, it seems that everything works well. But what if we add a new context? For example, many interpreters need to pre-define some primitive functions, which are called *intrinsics*. Therefore, we should add an intrinsic environment to store these intrinsic functions. Just like Common Lisp, functions and values do not share the same namespace in this expression language, so these two environments are independent of each other. We need a second parameter for `eval`, which requires modifying all existing code again:

```
type Eval = { eval : EnvN -> EnvF -> Int };
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval (envN:EnvN) (envF:EnvF) = n;
  (Add e1 e2).eval (envN:EnvN) (envF:EnvF) = e1.eval envN envF + e2.eval envN envF;
};
evalVar = trait implements VarSig<Eval> => {
  (Let s e1 e2).eval (envN:EnvN) (envF:EnvF) =
    e2.eval (insert @Int s (e1.eval envN envF) envN) envF;
  (Var       s).eval (envN:EnvN) (envF:EnvF) = lookup @Int s envN;
};
evalFunc = trait implements FuncSig<Eval> => {
  (LetF s f e).eval (envN:EnvN) (envF:EnvF) = e.eval envN (insert @Func s f envF);
  (AppF  s  e).eval (envN:EnvN) (envF:EnvF) = (lookup @Func s envF) (e.eval envN
    envF);
};
```

Such an approach to adding contextual information has two main problems:

- **It is highly non-modular:** Every time a new context is needed, all the existing code has to be modified. What is worse, we cannot easily modify the type of context if the previous definitions are from a library. In other words, the library author has to anticipate what kind of contexts will be needed in the future, which is impossible.

- **It does not encapsulate contexts:** Interfaces of contexts are fully exposed, even if they are not directly used. For example, `Let` and `Var` do not touch `envF`, while `LetF` and `AppF` do not touch `envD`. Such unnecessary exposure may lead to unexpected modifications to the contexts which ought to be hidden to avoid exploitation by malicious code.

5.3.2 Polymorphic Contexts

To address the two problems identified in the previous section, we propose a technique that relies on *disjoint polymorphism*, *intersection types* and *nested composition*. This technique enables modular, encapsulated contexts for modular components. Since we cannot anticipate what a context will evolve to in the future, our idea is to make contexts subject to change using parametric polymorphism.

Evaluation with a polymorphic context Instead of creating an interface for evaluation with specific contexts, we can parametrize the type of the context:

```
type Eval Context = { eval : Context -> Int };
```

Here `Context` is a type parameter. In essence `Eval` becomes a variant of the *Reader* Monad [Wadler, 1992], which is commonly used in functional programming. Similarly to Monads, some initial planning is necessary to adapt the existing code to use polymorphic contexts:

```
evalNum Context = trait implements ExpSig<Eval Context> => {
  (Lit     n).eval (ctx:Context) = n;
  (Add e1 e2).eval (ctx:Context) = e1.eval ctx + e2.eval ctx;
};
```

The trait `evalNum` has a type parameter `Context`, which is used as the type of the context in evaluation. Importantly, when implementing `evalNum`, the only thing one can do with `ctx` is to pass it to the children's call on `eval` since `ctx` is the only value of type `Context` in scope. In other words, parametric polymorphism enforces the encapsulation of the context and ensures that the correct context is passed to the evaluation of the children. To illustrate how CP enforces encapsulation of the context, suppose that we try to extract information from the polymorphic context, for example, by trying to look up a variable in the context:

```
evalNum Context = trait implements ExpSig<Eval Context> => {
  (Lit n).eval (ctx:Context) = lookup @Int "foobar" ctx;  -- Type Error!
  -- (Add e1 e2).eval ... is omitted
}
```

This code fails to type check because the type of `ctx` (i.e. `Context`) is not a subtype of `EnvN`, and there is no dynamic casting in CP that can change its type to `EnvN`. In short, if the polymorphic context is completely polymorphic (i.e. its type is just a type variable) then there is not much that can be done with the context except passing it down to recursive calls.

To instantiate `evalNum`, which requires no context on its own, we can specify `Context` as `Top` and pass `()` (the canonical top value) to `eval`:

```
(new evalNum @Top ,, expAdd @(Eval Top)).add.eval ()  --> 12
```

While the code requires an initial modification to be adapted to polymorphic contexts, no additional changes are necessary when future contexts are added to the program. Moreover, a nice quality of the code is that it is written in a direct style. Using more sophisticated solutions, such as Monads, would give additional expressive power, but often at the cost of writing code in a different style (for instance, with the monadic `do` notation).

Adding components with contexts Let us revisit the variable binding example. In order to add constructs that deal with variables and binders, a context with an `envD` field is needed:

```
    e.eval ({ envF = insert @Func s f ctx.envF } ,, ctx:Context);
```

Now, we have to write the code for a trait that deals with the evaluation of variables and binders. But what should be the type of context in this case? Using a fully polymorphic context, as we did for literal and addition expressions, will not work, because we need to extract and update information from the environment. Furthermore, using the type `CtxN` directly as the type of the environment is too specific, because it forces the contexts to contain exactly `CtxN` and nothing else. This would prevent modular context evolution. The answer is to use a context with the intersection type `CtxN&Context`:

```
evalVar (Context * CtxN) = trait implements VarSig<Eval (CtxN&Context)> => {
  (Let s e1 e2).eval (ctx:CtxN&Context) =
    e2.eval ({ envN = insert @Int s (e1.eval ctx) ctx.envN } ,, ctx:Context);
  (Var       s).eval (ctx:CtxN&Context) = lookup @Int s ctx.envN;
};
```

`Context` is a type variable *disjoint* with `CtxN` (expressed as the annotation `Context * CtxN`). This is an example of disjoint polymorphism [Alpuim et al., 2017], which is supported in CP. The disjointness constraint ensures that when the type variable is instantiated to a concrete type, that type cannot share a common supertype with `CtxN`. By using the type `CtxN&Context` as the type of context, we ensure that we can access the environment while being oblivious of other information in the context. Thus the context remains partly polymorphic and adaptable to future extensions while retaining encapsulation and ensuring that the other information in `Context` cannot be altered.

A record update problem The variable environment should be updated during the evaluation of the `Let` expression, whereas any other information in the context should be retained as well. Note that, the type of `{envD = insert ...}` is `CtxN`, which does not match `CtxN&Context`. So we have to merge it with `(ctx:Context)` to get back the `Context` part. This upcasting is possible because `Context` is a supertype of `CtxN&Context`. The disjoint constraint also ensures that the merge passes type-checking. The code illustrates that in CP we can do a *polymorphic record update* [Cardelli and Mitchell, 1991], which is a notorious problem in many calculi with polymorphic records. For instance, it is well-known that $F_{<:}$ with records (and many other calculi with bounded quantification) cannot solve the polymorphic record update problem. There are only a few calculi with polymorphic records and subtyping that can deal with this problem [Cardelli and Mitchell, 1991; Cardelli, 1994; Poll, 1997]. CP and its core foundation F_i^+ are among them.

A second component with a context To support intrinsic functions, we need a second environment `EnvF`, and a corresponding trait family:

```
type CtxF = { envF : EnvF };
evalFunc (Context * CtxF) = trait implements FuncSig<Eval (CtxF&Context)> => {
  (LetF s f e).eval (ctx:CtxF&Context) =
    e.eval ({ envF = insert @Func s f ctx.envF } ,, ctx:Context);
  (AppF  s  e).eval (ctx:CtxF&Context) = (lookup @Func s ctx.envF) (e.eval ctx);
};
```

Similarly, a polymorphic record update is needed for the `LetF` expression. Although these polymorphic trait families are defined separately, we can still merge them together using nested composition:

```
expPoly Exp = trait [self : ExpSig<Exp>&VarSig<Exp>&FuncSig<Exp>] => {
  test = new LetF "f" (\(x:Int) -> x * x)
                    (new Let "x" (new Lit 9) (new AppF "f" (new Var "x")));
};
e = new evalNum @(CtxN&CtxF) ,, evalVar @CtxF ,, evalFunc @CtxN ,,
        expPoly @(Eval (CtxN&CtxF));
e.test.eval { envN = empty @Int, envF = empty @Func }  --> 81
```

During the composition of `e`, the context types used by other trait families are passed as type arguments to make the final context consistent.

5.3.3 L-Attributed Grammars

The previous examples only access attributes from parents. It is also possible for inherited attributes to depend on siblings. There is a superset of the aforementioned

S-attributed grammars called *L-attributed grammars* [Knuth, 1968], where inherited attributes depend on parents and left siblings. In such grammars, attributes can be easily evaluated by a left-to-right depth-first traversal.

To illustrate L-attributed grammars, we take pretty-printing as an example. We use a position number to represent each terminal node in this pretty-printing function. The position number is determined by the pre-order traversal of the syntax tree. Before computing it, we need an auxiliary attribute called cnt that calculates the total number of nodes in the current subtree:

```
type Cnt = { cnt : Int };
cnt = trait implements ExpSig<Cnt> => {
  (Lit     n).cnt = 1;
  (Add e1 e2).cnt = e1.cnt + e2.cnt + 1;
};
```

With the help of cnt, we can compute that $e_1.\texttt{pos} = e_0.\texttt{pos}+1$ and $e_2.\texttt{pos} = e_0.\texttt{pos}+e_1.\texttt{cnt}+1$, where e_0 is the parent node of e_1 and e_2. The latter equation is a typical example of L-attributed grammars because the attribute depends on both its parent (e_0.pos) and left sibling (e_1.cnt). In our encoding, such computation is done on the parent side and the result is passed down using polymorphic contexts:

```
type Pos = { pos : Int };
type InhPos = { pos1 : Pos -> Int; pos2 : Pos -> Cnt -> Int };
type PrintPos Ctx = { print : Pos&Ctx -> String };
printPos (Ctx * Pos) = trait [self : InhPos] implements ExpSig<Cnt % PrintPos Ctx> =>
    {
  (Lit     n).print (inh:Pos&Ctx) = "{" ++ inh.pos.toString ++ "}";
  (Add e1 e2).print (inh:Pos&Ctx) =
    "(" ++ e1.print ({ pos = pos1 inh } ,, inh:Ctx) ++ "+" ++
           e2.print ({ pos = pos2 inh e1 } ,, inh:Ctx) ++ ")";
  pos1 (e0:Pos) = e0.pos + 1;
  pos2 (e0:Pos) (e1:Cnt) = e0.pos + e1.cnt + 1;
};
```

To compute the inherited attribute pos, we introduce two auxiliary functions in InhPos. They are implemented in accordance with the equations stated before. The self-type annotation [self:InhPos] is added for (Add e1 e2).print to access these two functions. Just like previous environments in interpreters, pos serves as a part of the context parameter of print. Since there is a child dependency on cnt, the sort is instantiated as <Cnt % PrintPos Ctx>. The position number is calculated and passed down via print calls in Add, while inh.pos is finally used in Lit.

With all of the traits ready, we can compose an expression and call the print function like before:

```
exp Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Add (new Lit 1) (new Lit 2)) (new Add (new Lit 3) (new Lit 4));
};
e = new cnt ,, printPos @Top ,, exp @(Cnt & PrintPos Top);
e.test.print { pos = 0 }  --> (({2}+{3})+({5}+{6}))
```

Since no other contexts need to be mixed together, we just pass Top as the type argument of printPos. In the last invocation on print, we set the initial context to {pos = 0}. This means that the root node of the whole syntax tree is marked as 0, then the parent of the left subtree is 1 and the leaves are 2 and 3. Similarly, the parent of the right subtree is 4 and the leaves are 5 and 6. We finally print the position numbers of leaf nodes, as the code shows above.

In fact, our encoding does not only allow L-attributed grammars, any inherited and synthesized attributes can be implemented by contextual evaluation. Nevertheless, L-attributed grammars correspond to a one-pass traversal and ensure termination, so they are the most common usage of attribute grammars and a good example for polymorphic contexts.

Summary With polymorphic contexts, L-attribute-grammar-like programs are expressed in a statically safe way. Such programs are not uncommon in real-world applications. Interpreters or other operations over ASTs are typical applications where non-trivial forms of attributes can occur. We have shown how variable and intrinsic environments are supported in the previous examples. Besides the two, more contexts may emerge when a language evolves, such as dynamic scoping, mutable parameters, and error handling.

There are three advantages of our approach to polymorphic contexts: 1) it enforces the recursive calls to take the full context as an argument; 2) the polymorphic portion of the context cannot be fiddled with, i.e., the only thing one can do is to pass it unchanged; 3) nonetheless, polymorphic contexts can still be refined for particular uses and expose just the right amount of information while hiding the remaining information. Polymorphic contexts, as well as the ability to express complex forms of attributes, are a valuable supplement to the modularity of Compositional Programming.

5.4 Formalization

This section presents the syntax and semantics of CP. The syntax of CP extends SEDEL [Bi and Oliveira, 2018] with constructs for Compositional Programming (i.e. compositional interfaces, compositional traits, method patterns, and nested trait composition). The semantics of CP is given by elaborating to a call-by-name formulation of F_i^+ [Bi et al., 2019], a typed calculus combining disjoint intersection types and polymorphism with BCD-style subtyping [Barendregt et al., 1983]. Nested trait composition is built on top of F_i^+'s nested composition [Bi et al., 2018]. We prove that the elaboration to F_i^+ is type-safe and coherent.

5.4.1 Syntax

Figure 5.1 gives the syntax of CP. A program is a sequence of declarations followed by an expression.

Declarations There are two kinds of declarations: type declarations and term declarations. A type declaration **type** $X\langle\overline{\alpha}\rangle$ **extends** $A = B$ is used for two purposes: introducing type aliases and declaring compositional interfaces. To simplify the formalization, we

$$
\begin{array}{llcl}
\text{Program} & P & ::= & D; P \mid E \\
\text{Declarations} & D & ::= & M \mid \textbf{type}\ X\langle\overline{a}\rangle\ \textbf{extends}\ A = B \\
\text{Term declarations} & M & ::= & x = E \mid (L\ \overline{x : A}\ [\textbf{self} : B]).l = E \\
\text{Types} & A, B & ::= & \textbf{Int} \mid a \mid \top \mid \bot \mid A \to B \mid \forall (a * A).B \mid A \,\&\, B \mid \{l : A\} \\
 & & \mid & \textbf{Trait}[A, B] \mid X\langle\overline{S}\rangle \\
\text{Sorts} & S & ::= & A \mid A \,\%\, B \\
\text{Expressions} & E & ::= & i \mid x \mid \top \mid \lambda x.E \mid E_1\ E_2 \mid \Lambda(a * A).E \mid E\ @A \mid E_1\ ,,\ E_2 \mid \{\overline{M}\} \mid E.l \\
 & & \mid & E : A \mid \textbf{let}\ x : A = E_1\ \textbf{in}\ E_2 \mid \textbf{open}\ E_1\ \textbf{in}\ E_2 \mid \textbf{new}\ E \mid E_1 \hat{}\, E_2 \\
 & & \mid & \textbf{trait}[\textbf{self} : A]\ \textbf{implements}\ B\ \textbf{inherits}\ E_1\ \texttt{=>}\ E_2 \\
\text{Term contexts} & \Gamma & ::= & \bullet \mid \Gamma, x : A \\
\text{Type contexts} & \Delta & ::= & \bullet \mid \Delta, a * A \mid \Delta, X\langle\overline{a}, \overline{\beta}\rangle \mapsto A \\
\text{Sort contexts} & \Sigma & ::= & \bullet \mid \Sigma, a \mapsto \beta
\end{array}
$$

Figure 5.1: *Syntax of CP.*

do not formalize type declarations with conventional type parameters (i.e. we only consider type declarations with sorts). However, adding type parameters can be done in standard ways and we have such declarations in our implementation. The type (or type constructor) X is parametrized with a sequence of sorts $\langle\overline{a}\rangle$ and can optionally extend another type. There are two forms of term declarations: $x = E$ is for simple variable bindings; $(L\ \overline{x : A}\ [\textbf{self} : B]).l = E$ is a *method pattern* serving as syntactic sugar for defining single-field traits conveniently.

Types Metavariables A and B range over types. Types include integers **Int**, type variables a, the top type $\top$, the bottom type $\bot$, arrows $A \to B$, disjoint quantification $\forall(a * A).B$, intersections $A \,\&\, B$, single-field record types $\{l : A\}$ (multi-field record types are syntax sugar for intersections of multiple single-field record types) , trait types $\textbf{Trait}[A, B]$, and type aliases with sorts instantiated $X\langle\overline{S}\rangle$, where each sort can be either instantiated using a type or a pair of types.

Expressions Metavariable E ranges over expressions. Expressions include integer literals i, term variables x, the top value $\top$, lambda abstractions $\lambda x.E$, term applications $E_1\ E_2$, type abstractions $\Lambda(a * A).E$, type applications $E\ @A$, merges $E_1\ ,,\ E_2$, multi-field records $\{\overline{M}\}$, record projections $E.l$, annotated expressions $E : A$, and (recursive) let bindings $\textbf{let}\ x : A = E_1\ \textbf{in}\ E_2$. There are also a few trait related constructs. $\textbf{trait}[\textbf{self} : A]\ \textbf{implements}\ B\ \textbf{inherits}\ E_1\ \texttt{=>}\ E_2$ specifies a an explicit **self** reference of type A, type B to implement, an explicit **self** reference of type A, an inherited trait expression E_1 and a body expression E_2. The **new** E construct instantiates a trait expression. $E_1 \hat{}\, E_2$ is the forwarding expression inherited from SEDEL [Bi and Oliveira, 2018]. Inspired by ML-like modules [MacQueen, 1984], **open** E_1 **in** E_2 is a new construct for directly accessing fields from a record without explicit projections.

5.4.2 An Informal Introduction to the Elaboration

The semantics of CP is defined by elaborating to F_i^+ extended with recursive let bindings. Elaborating a CP program into a F_i^+ expression takes several steps, including desugaring, sort transformation, type expansion, etc. The elaboration builds on two ideas from the literature: the denotational model of inheritance by Cook and Palsberg [1989], and generalized Object Algebras [Oliveira et al., 2013]. To better understand the elaboration, let us revisit some of the examples presented in Section 5.2 and show their concrete elaborations into CP code using more atomic features, such as record types and records, and recursive let bindings.

Elaborating compositional interfaces and sorts. Firstly, the compositional interface `ExpSig`:

```
type ExpSig<Exp> = {
  Lit : Int -> Exp;
  Add : Exp -> Exp -> Exp;
};
```

is translated to an equivalent type using only type parameters, and the sort `Exp` is eliminated:

```
type ExpSig Exp OExp =
  { Lit : Double -> Trait[Exp,OExp] } &
  { Add : Exp -> Exp -> Trait[Exp,OExp] };
```

The sort `Exp` is represented by two type parameters, `Exp` and `OExp`, for respectively capturing the negative and positive occurrences of `Exp`. Negative occurrences of `Exp` (at input positions) are kept unchanged while positive occurrences of `Exp` (at output positions) are changed to `OExp`. For `Add`, the two parameters of type `Exp` are negative. Therefore, only the return type of `Add` is transformed. Since `Add` is a constructor, positive occurrences of `Exp` are further translated to a trait type `Trait[Exp,OExp]`. As a type synonym, `ExpSig` and its right-hand side are put into the type context that tracks type declarations, among other things. In essence, this transformation is inspired by generalized Object Algebra interfaces [Oliveira et al., 2013] (see also 2.4.1), and the distinction of positive and negative occurrences is helpful for expressing dependencies. If we just wanted to model modular programs *without dependencies*, then distinguishing between positive and negative occurrences would not be necessary.

Similarly, the extended compositional interface `MulSig<Exp>`:

```
type MulSig<Exp> extends ExpSig<Exp> = {
  Mul : Exp -> Exp -> Exp;
};
```

is translated to a type equivalent to:

```
type MulSig Exp OExp =
  { Lit : Double -> Trait[Exp,OExp] } &
  { Add : Exp -> Exp -> Trait[Exp,OExp] } &
  { Mul : Exp -> Exp -> Trait[Exp,OExp] };
```

where the type appearing in the extends clause is expanded by looking up the type context and intersecting that type with the type on the original right-hand side.

Elaborating traits. The trait family evalNum implements ExpSig:

```
evalNum = trait implements ExpSig<Eval> => {
  (Lit     n).eval = n;
  (Add e1 e2).eval = e1.eval + e2.eval;
};
```

which is desugared to:

```
evalNum = trait [self: Top] implements ExpSig Eval Eval => open self in
  { Lit = \(n: Double) -> trait => { eval = n } } ,,
  { Add = \(e1: Eval) -> \(e2: Eval) -> trait => { eval = e1.eval + e2.eval} };
```

The sort instantiation <Eval> indicates that there are no dependencies. Therefore, both type parameters (Exp and OExp) in the elaborated code are instantiated as Eval. Since no self-type is specified, the self-reference has type Top by default. We can ignore the open expression for the moment, since it has no effect in this case, as the self type is Top. Method patterns are desugared to functions returning traits and multi-field records are desugared to a merge of multiple single-field records. After type-checking, evalNum is further elaborated into a F_i^+ expression:

```
let evalNum = \(self: Top) ->
  { Lit = \(n: Int) -> \(self: Top) -> { eval = n } } ,,
  { Add = \(e1: Eval) -> \(e2: Eval) -> \(self: Top) -> { eval = e1.eval + e2.eval } }
in ...
```

The term declaration is elaborated to a let expression and the trait expressions are elaborated to functions. Since no self-types are specified, the self parameters are all of type Top.

Elaborating child dependencies. printChild also implements ExpSig but contains child dependencies:

```
printChild = trait implements ExpSig<Eval % Print> => {
  (Lit     n).print = n.toString;
  (Add e1 e2).print = if e2.eval == 0 then e1.print
                      else "(" ++ e1.print ++ "+" ++ e2.print ++ ")";
};
```

This trait is desugared to:

```
printChild = trait [self: Top] implements ExpSig (Eval&Print) Print => open self in
  { Lit (n: Double) = trait => { print = n.toString } } ,,
  { Add (e1: Eval&Print) (e2: Eval&Print) =
    { print = if e2.eval == 0 then e1.print
              else "(" ++ e1.print ++ "+" ++ e2.print ++ ")" } };
```

Since printChild instantiates the sort as <Eval % Print>, Exp and OExp are respectively instantiated as Eval&Print (i.e. the intersection of the two types in <Eval % Print>) and Print (i.e. the second type in <Eval % Print>). Through some type inference, the arguments with expression types in the constructors (such as e1 and e2) are of type Eval&Print. This enables eval to be called on e1 and e2 without an implementation of that operation in the current trait. In short, printChild is elaborated to an expression similar to evalNum except that expressions arguments are of different type (Eval&Print) from the output. Importantly, for the modular definition of printChild to work it is crucial to use different types in the instantiation of Exp for input (negative) positions and output (positive) positions.

Elaborating self-type annotations On the other hand, traits with explicit self-type annotations are desugared differently. For instance, expAdd:

```
expAdd Exp = trait [self : ExpSig<Exp>] => {
  test = new Add (new Lit 4) (new Lit 8);
};
```

is desugared to:

```
expAdd = /\Exp. trait [self: ExpSig Exp Exp] => open self in {
  test = new Add (new Lit 4) (new Lit 8);
};
```

The original trait body is wrapped into an open expression so that the constructors/methods exposed by the self-type are directly in scope. Further elaboration into F_i^+ results in:

```
let expAdd =
  /\Exp. \(self : { Lit : Int -> Exp -> Exp } & { Add : Exp -> Exp -> Exp -> Exp }) ->
    let Add = self.Add
    in let Lit = self.Lit
    in { test = letrec self : Exp = Add (letrec self : Exp = Lit 4 self in self)
                                        (letrec self : Exp = Lit 8 self in self)
                                        self
                in self }
in ...
```

The type alias used in specifying the self-type is expanded. The type translation rewrites trait types as function types. The open self expression is elaborated into a series of let bindings, one for each record label. The new expressions are elaborated to lazy fixed-point of self, following Cook and Palsberg denotational model of inheritance [Cook and Palsberg, 1989].

Elaborating inheritance and overriding We now show the elaboration of a trait that additionally uses inherits and override. An instance is expMul:

```
expMul Exp = trait [self : MulSig<Exp>] inherits expAdd @Exp => {
  override test = new Mul super.test (new Lit 4);
};
```

which is elaborated as:

```
let expMul = /\ Exp. \(self : { Lit : Int -> Exp -> Exp } &
                              { Add : Exp -> Exp -> Exp -> Exp } &
                              { Mul : Exp -> Exp -> Exp -> Exp }) ->
  let super = (expAdd Exp) self
  in (super : Top) ,,
    let Add = self.Add
    in let Lit = self.Lit
    in let Mul = self.Mul
    in { test = letrec self : Exp = Mul super.test
                                        (letrec self : Exp = Lit 4 self in self)
                                        self
                in self }
in ...
```

The inherited trait expression (`expAdd`) is elaborated as a function and applied to the self-reference and the result is bound to **`super`**. Then **`super`** is merged with the body of the trait. Since `test` is overridden, it should be excluded from super; otherwise a conflict will occur when merging super with the body. The exclusion of `test` from **`super`** is done by injecting a type annotation `Top`.

Elaborating nested composition of traits Finally, we show how the merge of traits is elaborated using `(new evalNum ,, expAdd @Eval).test.eval` as an example:

```
letrec oself : { Lit : Int -> Top -> {eval : Int} } &
               { Add : {eval : Int} -> {eval : Int} -> Top -> {eval : Int} } &
               { test : {eval : Int} } =
  \(iself : { Lit : Int -> {eval : Int} -> {eval : Int} } &
            { Add : {eval : Int} -> {eval : Int} -> {eval : Int} -> {eval : Int} })
    -> (evalNum iself) ,, ((expAdd {eval : Int} iself) oself)
in oself.test.eval
```

where we have renamed the outer and inner self as `oself` and `iself` for distinction. `evalNum ,, expAdd @Eval` is elaborated to a function that returns the merge of respectively applying `evalNum` and `expAdd @Eval`, which are already elaborated as functions, to `iself`. Since the merged trait is of type `Trait[ExpSig<Eval>,ExpSig<Eval>&{test:Eval}]`, `iself` has the elaborated type of `ExpSig<Eval>` while `oself` has the elaborated type of `ExpSig<Eval>&{test:Eval}`.

5.4.3 Static Semantics

Overview Figure 5.2 gives an overview of all the relations involved in the elaboration, where $\rightarrow$ denotes the transformation flow and $\dashrightarrow$ denotes the dependencies between the relations. Firstly, patterns and multi-field records are desugared by $[\![\cdot]\!]$. Then the program P is elaborated into a F_i^+ expression e by $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$. During the elaboration process, some transformations and checks are performed: type synonyms and compositional interfaces $X\langle\overline{S}\rangle$ are expanded by $\Delta, \Sigma \vdash A \Rightarrow B$; the right-hand side of a type declaration is transformed by $\Sigma \vdash_P^c A \Rightarrow B$ for distinguishing positive and negative occurrences of sorts; the subtyping relation between two types is checked by $A <: B$; the disjointness of two types is ensured by $\Delta \vdash A * B$. Both subtyping and disjointness checking rely on top-like types $(\rceil A \lceil)$. The latter further relies on disjointness axioms $(A *_{ax} B)$.

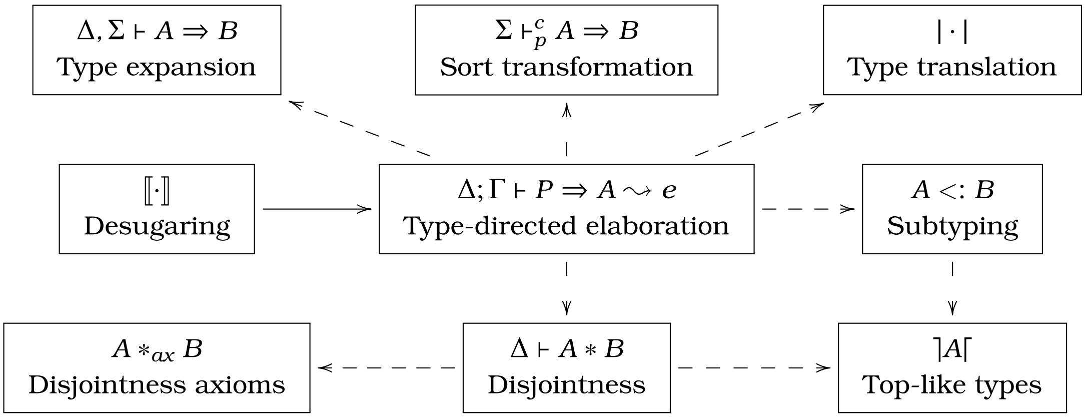


Figure 5.2: *The relation of relations.*

Desugaring The core desugaring rules are:

$$
\begin{aligned}
&[\![\{M_1, \dots, M_n\}]\!] && \overset{\text{def}}{=} \quad [\![\{M_1\}]\!] \;,,\; \dots \;,,\; [\![\{M_n\}]\!] \\
&[\![(L\ \overline{x : A}\ [\mathbf{self} : B]).l = E]\!] && \overset{\text{def}}{=} \\
&\quad L = \lambda\overline{x : A}.[\![\mathbf{trait}[\mathbf{self} : B]\ \mathbf{implements}\ \top\ \mathbf{inherits}\ \top\ \mathbf{=>}\ \{l = E\}]\!] && \\
&[\![\mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ E_1\ \mathbf{=>}\ E_2]\!] && \overset{\text{def}}{=} \\
&\quad \mathbf{trait}[\mathbf{self} : A]\ \mathbf{implements}\ B\ \mathbf{inherits}\ [\![E_1]\!]\ \mathbf{=>}\ \mathbf{open\ self\ in}\ [\![E_2]\!] &&
\end{aligned}
$$

The first rule desugars a multi-field record into an intersection of singleton records. The second rule desugars a method pattern into an ordinary variable binding to a function that returns a single-field trait. The last rule implicitly opens `self` for the trait body so that members declared by the self-type can be directly accessed without prefixing `self`.

Type checking Figure 5.3 shows the selected typing rules. The gray parts could be ignored for the moment. They will be discussed later in Section 5.4.4. The type system of CP is bidirectional: under the contexts Δ and Γ, the inference mode ($\Rightarrow$) synthesizes a type A while the checking mode ($\Leftarrow$) checks against A. A lot of the rules are presented previously in the literature [Bi et al., 2019], thus we discuss only the novel ones, which mostly relate to traits and declarations.

The rule T-TYDECL adds a type alias to the type context Δ. The right-hand side, type A, is expanded and transformed before it is added to Δ for type-checking the remaining program. Each sort α has a fresh companion variable β (corresponding to `OExp` in the examples) for distinguishing negative and positive occurrences of sorts. The rule T-TMDECL deals with term declarations. The bound expression, E, is inferred with a type A. Then, x of type A is added to the term context Γ for type-checking the remaining program.

The rule T-TRAIT is the most complicated one since multiple types and expressions are involved and several validity checks are performed. It firstly expands the self type A and the type to implement B as A_1 and B_1. Then **self** is added to Γ in type-checking the inherited expression E_1 and the body E_2. The inherited expression E_1 is valid only when it is of a trait type **Trait**$[A_2, B_2]$ and the requirement A_2 is met by A_1. After validity checking, **super** can be added to Γ in type-checking the body E_2. The type of body should be disjoint

$\boxed{\Delta;\Gamma \vdash P \Rightarrow A \rightsquigarrow e}$

T-tyDecl

$$\frac{\Delta;\overline{\alpha \mapsto \beta} \vdash B \Rightarrow B_1 \qquad \dfrac{\text{fresh } \overline{\beta} \qquad \Delta;\overline{\alpha \mapsto \beta} \vdash A \Rightarrow A_1}{\overline{\alpha \mapsto \beta} \vdash^{\text{false}}_{+} B_1 \Rightarrow B_2} \qquad \Delta, X\langle\overline{\alpha},\overline{\beta}\rangle \mapsto A_1 \,\&\, B_2;\Gamma \vdash P \Rightarrow C \rightsquigarrow e}{\Delta;\Gamma \vdash \textbf{type } X\langle\overline{\alpha}\rangle \textbf{ extends } A = B; P \Rightarrow C \rightsquigarrow e}$$

T-tmDecl

$$\frac{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e_1 \qquad \Delta;\Gamma, x : A \vdash P \Rightarrow B \rightsquigarrow e_2}{\Delta;\Gamma \vdash x = E; P \Rightarrow B \rightsquigarrow \textbf{let } x : |A| = e_1 \textbf{ in } e_2}$$

$\boxed{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e}$ (Inference)

T-trait

$$\frac{\begin{array}{c}\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta;\bullet \vdash B \Rightarrow B_1 \qquad \Delta;\Gamma, \textbf{self} : A_1 \vdash E_1 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_1 \\ A_1 <: A_2 \qquad \Delta;\Gamma, \textbf{self} : A_1, \textbf{super} : B_2 \vdash E_2 \Rightarrow C \rightsquigarrow e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash \textbf{trait}[\textbf{self} : A] \textbf{ implements } B \textbf{ inherits } E_1 \textbf{ => } E_2 \Rightarrow \textbf{Trait}[A_1, C \,\&\, B_2] \\ \rightsquigarrow \lambda(\textbf{self} : |A_1|).\, \textbf{let super} = e_1 \textbf{ self in } e_2 \,,,\, \textbf{super}\end{array}}$$

T-mergeTrait

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \textbf{Trait}[A_1, B_1] \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2}{\Delta;\Gamma \vdash E_1 \,,,\, E_2 \Rightarrow \textbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2] \rightsquigarrow \lambda(\textbf{self} : |A_1 \,\&\, A_2|).\, e_1 \textbf{ self} \,,,\, e_2 \textbf{ self}}$$

T-new

$$\frac{\Delta;\Gamma \vdash E \Rightarrow \textbf{Trait}[A, B] \rightsquigarrow e \qquad B <: A}{\Delta;\Gamma \vdash \textbf{new } E \Rightarrow B \rightsquigarrow \textbf{let self} : |B| = e \textbf{ self in self}}$$

T-open

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \overline{\{l_i : A_i\}} \rightsquigarrow e_1 \qquad \Delta;\Gamma, \overline{l_i : A_i} \vdash E_2 \Rightarrow e_2 \rightsquigarrow B}{\Delta;\Gamma \vdash \textbf{open } E_1 \textbf{ in } E_2 \Rightarrow B \rightsquigarrow \overline{\textbf{let } l_i : |A| = e_1.l_i} \textbf{ in } e_2}$$

Figure 5.3: *Selected typing rules.*

to the type of the inherited expression E_1 ($C * B_2$). Meanwhile the body and the inherited expression should together implement what B_1 specifies ($C \,\&\, B_2 <: B_1$). The rule T-new instantiates a trait. The expression E should be of type $\textbf{Trait}[A, B]$ and the requirement A should be met by B. The rule T-open collects the record types from the inferred type of E_1 and adds every label type pair to the term context in inferring E_2. Besides a rule for ordinary merges, T-mergeTrait is a novel rule specially for the merge of two traits. T-mergeTrait says that if the two expressions are of type $\textbf{Trait}[A_1, B_1]$ and $\textbf{Trait}[A_2, B_2]$ and B_1 and B_2 are disjoint, then the merged expression is of type $\textbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2]$. Inferring the merged traits as a trait type rather than an intersection type brings several advantages, which will be discussed in Chapter 6.

Sort transformation Figure 5.4 shows the sort transformation, which is a phase that replaces positive occurrences of a sort appearing in a type declaration with some other type. Sort transformation is illustrated in the previous section by elaborating the right-hand side of `ExpSig` and `MulSig`. From the names we cannot distinguish sorts from type

$\boxed{\Sigma \vdash^{c}_{p} A \Rightarrow B}$

$$\textsc{TR-positive}\quad \frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash^{\text{false}}_{+} \alpha \Rightarrow \beta} \qquad \textsc{TR-ctrPositive}\quad \frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash^{\text{true}}_{+} \alpha \Rightarrow \mathbf{Trait}[\alpha, \beta]} \qquad \textsc{TR-rcd}\quad \frac{\Sigma \vdash^{\textsc{isCapitalized}(l)}_{p} A \Rightarrow B}{\Sigma \vdash^{c}_{p} \{l : A\} \Rightarrow \{l : B\}}$$

$$\textsc{TR-arr}\quad \frac{\Sigma \vdash^{c}_{\text{flip}(p)} A \Rightarrow A_1 \qquad \Sigma \vdash^{c}_{p} B \Rightarrow B_1}{\Sigma \vdash^{c}_{p} A \rightarrow B \Rightarrow A_1 \rightarrow B_1} \qquad \textsc{TR-trait}\quad \frac{\Sigma \vdash^{c}_{\text{flip}(p)} A \Rightarrow A_1 \qquad \Sigma \vdash^{c}_{p} B \Rightarrow B_1}{\Sigma \vdash^{c}_{p} \mathbf{Trait}[A, B] \Rightarrow \mathbf{Trait}[A_1, B_1]}$$

$\boxed{\Delta, \Sigma \vdash A \Rightarrow B}$

$$\textsc{E-tvar}\quad \frac{\alpha * A \in \Delta}{\Delta, \Sigma \vdash \alpha \Rightarrow \alpha} \qquad \textsc{E-sig}\quad \frac{X\langle\overline{\alpha}, \overline{\beta}\rangle \mapsto C \in \Delta \qquad \overline{\Sigma \vdash S \Rightarrow \langle A, B\rangle}}{\Delta, \Sigma \vdash X\langle\overline{S}\rangle \Rightarrow [\overline{A/\alpha}, \overline{B/\beta}]C}$$

$\boxed{\Sigma \vdash S \Rightarrow \langle A, B\rangle}$

$$\textsc{E-sort1Sort}\quad \frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash \alpha \Rightarrow \langle\alpha, \beta\rangle} \qquad \textsc{E-sort1}\quad \Sigma \vdash A \Rightarrow \langle A, A\rangle \qquad \textsc{E-sort2}\quad \Sigma \vdash A\%B \Rightarrow \langle A \,\&\, B, B\rangle$$

Figure 5.4: *Selected sort transformation and type expansion rules.*

parameters and therefore a sort context Σ is needed. How a sort is transformed is further determined by two conditions: p and c. The condition p tracks the positions where sorts appear: $+$ indicates a positive position and $-$ indicates a negative position. Positive and negative positions are treated differently: negative positions are kept unchanged while positive positions are transformed. Specifically, for a function type $A \rightarrow B$, p is flipped in processing A but unchanged in processing B (rule TR-ARR). The same applies for $\mathbf{Trait}[A, B]$ (rule TR-TRAIT). The boolean value c bookmarks whether the type appears inside a constructor. By default false, c is set to be true when the label of a record is capitalized (rule TR-RCD). According to p and c, rules TR-POSITIVE and TR-CTRPOSITIVE transform the sort α to β and $\mathbf{Trait}[\alpha, \beta]$ respectively.

Type expansion Type expansion plays two roles: eliminating type aliases and ensuring the well-formedness of the types. It is used, for example, when elaborating the **extends** clause on `MulSig` and **implements** clauses on `evalNum` and `printChild`. Figure 5.4 also shows the relevant type expansion rules. The rule E-TVAR checks whether the type variable is in the type context. The rule E-SIG eliminates $X\langle\overline{S}\rangle$ by looking up X in the type context Δ and substituting the negative and positive occurrences of sorts (α and β). An instantiated sort S is expanded to a pair of types for substituting α and β respectively. There are three cases for expanding S: if S is a sort, then α and β are kept abstract (E-SORT1SORT); if S has only one type, both α and β are substituted by that type (E-SORT1); otherwise, α and β are substituted by the intersection of the two types ($A \,\&\, B$) and the second type (B) from the pair (E-SORT2).

$\boxed{A <: B}$

S-REFL
$$A <: A$$

S-TRANS
$$\frac{A <: B \qquad B <: C}{A <: C}$$

S-TOPLIKE
$$A <: \rceil B \lceil$$

S-BOT
$$\bot <: A$$

S-RCD
$$\frac{A <: B}{\{l : A\} <: \{l : B\}}$$

S-ANDL
$$A \,\&\, B <: A$$

S-ANDR
$$A \,\&\, B <: B$$

S-AND
$$\frac{A <: B \qquad A <: C}{A <: B \,\&\, C}$$

S-ARR
$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \rightarrow A_2 <: B_1 \rightarrow B_2}$$

S-FORALL
$$\frac{B_1 <: B_2 \qquad A_2 <: A_1}{\forall (a * A_1).B_1 <: \forall (a * A_2).B_2}$$

S-TRAIT
$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{\mathbf{Trait}[A_1, B_1] <: \mathbf{Trait}[A_2, B_2]}$$

S-DISTARR
$$(A \rightarrow B) \,\&\, (A \rightarrow C) <: A \rightarrow B \,\&\, C$$

S-DISTTRAIT
$$\mathbf{Trait}[A, B] \,\&\, \mathbf{Trait}[A, C] <: \mathbf{Trait}[A, B \,\&\, C]$$

S-DISTRCD
$$\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\}$$

S-DISTALL
$$\forall (a * A).B \,\&\, \forall (a * A).C <: \forall (a * A).B \,\&\, C$$

Figure 5.5: *Subtyping.*

$\boxed{\rceil A \lceil}$

TL-TOP
$$\rceil \top \lceil$$

TL-AND
$$\frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}$$

TL-ARR
$$\frac{\rceil B \lceil}{\rceil A \rightarrow B \lceil}$$

TL-RCD
$$\frac{\rceil A \lceil}{\rceil \{l : A\} \lceil}$$

TL-ALL
$$\frac{\rceil B \lceil}{\rceil \forall (a * A).B \lceil}$$

TL-TRAIT
$$\frac{\rceil B \lceil}{\rceil \mathbf{Trait}[A, B] \lceil}$$

Figure 5.6: *Top-like types*

Subtyping Figure 5.5 shows the subtyping rules. Most rules come from multiple sources in previous work [Barendregt et al., 1983; Bi and Oliveira, 2018; Bi et al., 2018, 2019]. The main novelty is the rule that deals with distributivity of trait types (rules S-DISTTRAIT),which essentially follows the TL-ARR rule for functions (which is inspired by BCD subtyping Barendregt et al. [1983]). The rule S-DISTTRAIT distributes intersections over the `Trait` type constructor. In the original work on first-class traits, the elaboration targeted the F_i calculus, which is a precursor of F_i^+ without distributivity rules. Thus, that work did not have distributivity for traits. The novel distributivity rules for traits are essential for achieving the nested composition of traits (and trait families). The trait rules and other distributive rules allow, for example, `ExpSig<Eval>&ExpSig<Print>` to be subtype of `ExpSig<Eval&Print>`. The rule S-TOPLIKE is also novel. It states that any type is a subtype of a top-like type.

Top-like types As discussed in Section 2.5, top-like types are types isomorphic to $\top$ (i.e. both sub- and supertypes of $\top$). We add a new rule TL-TRAIT for trait types, which states that a trait type is top-like when its provided interface is top-like.

Disjointness The disjointness judgment detects the conflicts when merging two expressions of type A and B. These rules are omitted here since they are merely a combination of the rules from F_i^+ (which can be found in Figure 2.4) and SEDEL. Interested readers can refer to Appendix A for the full disjointness rules of CP.

5.4.4 Formal Elaboration

The dynamic semantics of CP is given by an elaboration to F_i^+ (cf. Section 2.5), which is our target language. The semantics and metatheory (including type-safety and coherence) of F_i^+ have been studied in previous work [Bi et al., 2019]. The semantics of F_i^+ is given by elaborating to F_{co}, a variant of System F extended with products and explicit coercions. In the original paper by Bi et al. [2019], F_{co} has a call-by-value semantics. Since F_i^+ is define by elaboration to F_{co} it inherits the call-by-value semantics of F_{co}. Here we assume a call-by-name variant F_{co}, which we expect to be type-sound. We expect this result to be straightforward since F_{co} is just a minor variant of System F and System F is known to be type-sound in both call-by-value and call-by-name. The translations from CP to F_i^+ and then to F_{co} are unaffected by the choice of evaluation strategy in F_{co}, and simply inherit the evaluation strategy from F_{co}. As we have mentioned earlier, we also assume lazy recursive let bindings, which are not present in F_i^+. Lazy recursive let bindings are in fact the main motivation to switch to a call-by-name semantics, since they are necessary for the encodings of self references. Although we expect the proof of type-soundness of call-by-name F_{co} and coherence of call-by-name F_i^+ to easily hold, we leave such validation for future work.

F_i^+ is a subset of CP excluding declarations and trait related constructs. Therefore, elaborating the shared constructs is straightforward, and only elaborating constructs specific to CP requires some explanations. Let us focus on the the elaborated F_i^+ expressions (gray parts) shown in Figure 5.3. Intuitively, T-TmDecl elaborates a term declaration into a let expression, as illustrated by the elaborations on `evalNum`, `printChild` and `expAdd` in Section 5.4.2. The gray parts of T-Trait and T-New are inherited from SEDEL, which follows Cook and Palsberg [1989]'s denotational semantics of inheritance. Concretely, a trait expression is elaborated to a function that takes **self** as an argument, binds the application result of e_1 **self** to **super** and returns a merge of the body (e_2) and **super**. T-Trait is also illustrated by the elaborations on `evalNum`, `printChild` and `expAdd`. As shown by `expAdd`, T-New elaborates the expression into a lazy fixed-point of **self**. T-MergeTrait elaborates the merge of two traits into a function that takes **self** as an argument and returns the merge of applying the two elaborated traits e_1 and e_2 to **self**. Note that the type of the **self** argument for the merged trait is an intersection of the two self types from the two traits being merged. Elaboration on `evalNum ,, expAdd @Eval` illustrates T-MergeTrait. T-Open elaborates an **open** expression into a sequence of **let** expressions, one for each field from the record e_2. These **let** expressions bind labels to their projections so that e_2 can directly access all the fields from e_1 without explicit projections. The elaboration on `expAdd` is a showcase of T-Open.

Finally note that CP's types also need to be translated to F_i^+'s types, because in F_i^+

there is no **Trait**$[A, B]$ construct. All types, except **Trait**$[A, B]$ have straightforward translations. Trait types are translated to function types in F_i^+ (following SEDEL) with the rule $|\textbf{Trait}[A, B]| = |A| \rightarrow |B|$.

5.4.5 Type Safety and Coherence

The elaboration of CP into F_i^+ is type-safe and coherent. We summarize the key results here. Detailed proofs and other auxiliary lemmas can be found in Appendix B.

The first result is that elaboration is type-safe. To prove this result we need a few results for some of the auxiliary relations, which are shown next.

Lemma 5.1 *[Well-formedness preservation] If* $\Delta, \Sigma \vdash A \Rightarrow B$ *then* $|\Delta| \vdash |B|$.

Proof. By induction on the derivation of the judgment. □

Lemma 5.2 *[Disjointness axiom preservation] If* $A *_{ax} B$ *then* $|A| *_{ax} |B|$.

Proof. Note that $|\textbf{Trait}[A, B]| = |A| \rightarrow |B|$; the rest are immediate. □

Lemma 5.3 *[Subtyping preservation] If* $A <: B$ *then* $|A| <: |B|$.

Proof. By structural induction on the subtyping judgment. □

Lemma 5.4 *[Disjointness preservation] If* $\Delta \vdash A * B$ *then* $|\Delta| \vdash |A| * |B|$.

Proof. By structural induction on the disjointness judgment. □

Then the main type-safety theorem is:

Theorem 5.1 *[Type-safety] We have that:*

- *If* $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.

Proof. By structural induction on the typing judgment. □

The second theorem is the coherence of the elaboration:

Theorem 5.2 *[Coherence] Each well-typed CP program has a unique elaboration.*

Proof. For each elaboration rule, the elaborated F_i^+ expression in the conclusion is uniquely determined by the elaborated F_i^+ expressions in the premises. By the coherence property of F_i^+, we conclude that each well-typed CP program has a unique elaboration. Therefore CP is coherent. □

Additional Properties There are more properties proved in the F_i^+ paper, including the decidability of the type system. These properties should easily hold for CP by extending the proofs with a case for trait types. The cases for trait types are essentially similar to the cases of function types (trait types are actually encoded as function types in the elaboration to F_i^+), so the proof extensions should be straightforward. One thing to notice is that the subtyping relation presented in this paper is non-algorithmic due to the existence of a transitive rule (S-TRANS). An algorithmic variant of the subtyping relation is shown by Bi et al. [2019]. For proving decidability, we would need to use an extended version of such algorithmic subtyping with trait types.

5.5 Case Studies

To further demonstrate the applicability of CP, we conducted three case studies. The first case study is Scans, a small DSL for describing parallel circuits originally proposed by Hinze [2004] and further studied by Gibbons and Wu [2014]. The second one is a mini interpreter, which integrates and extends the examples in Section 5.2 and Section 5.3. The last case study is an implementation of the C0 compiler, inspired by the work of Rendel et al. [2014], which compiles a subset of C to JVM instructions. In all of the three case studies, the need for dealing with extensibility and complex dependencies arises.

5.5.1 Scans

Recall Scans introduced in Section 3.2.1. It is an interesting case study because implementing Scans modularly requires an approach not only to solving the EP but also capable of expressing dependencies. Besides the ones shown in Chapter 3, there are already a few implementations written in different languages [Gibbons and Wu, 2014; Zhang and Oliveira, 2019; Bi et al., 2019]. Still, these implementations are not fully satisfying. We compare our implementation with respect to these implementations. Note, however, that there are no dependencies on self-references in this case study. Thus this case study does not exercise such a form of dependencies.

Scans in CP The techniques shown in Section 5.2 are used in modularizing Scans with CP. The syntax of Scans is captured by a compositional interface:

```
type CircuitSig<Circuit> = {
  Identity : Int -> Circuit;
  Fan      : Int -> Circuit;
  Above    : Circuit -> Circuit -> Circuit;
  Beside   : Circuit -> Circuit -> Circuit;
  Stretch  : (List Int) -> Circuit -> Circuit;
};
```

Operations are modeled as trait families that concretely implement the compositional interface. For example, the implementation of `wellSized` is given below:

```
type WellSized = { wS : Bool };
wellSized = trait implements CircuitSig<Width,WellSized> => {
  (Identity   n).wS = true;
  (Fan        n).wS = true;
  (Above   c1 c2).wS = c1.wS && c2.wS && c1.width == c2.width;
  (Beside c1 c2).wS = c1.wS && c2.wS;
  (Stretch ns c).wS = c.wS && length ns == c.width;
};
```

The trait family `wellSized` implements `CircuitSig<Width%WellSized>`, indicating that it depends on another trait family of `CircuitSig` that implements `Width`. As discussed in Section 5.2.2, such dependency is weak and allows us to, for example, call `width` on `c` in `Stretch`.

New variants The new constructor RStretch is added by extending the compositional interface:

```
type NCircuitSig<Circuit> extends CircuitSig<Circuit> = {
  RStretch : (List Int) -> Circuit -> Circuit
};
```

Accordingly, existing trait families are extended with a case for RStretch, for example:

```
nWellSized = trait implements NCircuitSig<Width,WellSized> inherits wellSized => {
  (RStretch ns c).wS = c.wS && length ns == c.width;
};
```

Similarly, nWidth, nDepth, and nLayout extend their respective trait families.

Finally, we obtain the full implementation of Scans by composing all the operations as well as a generic trait that constructs a modular circuit:

```
scans = new nWidth,, nDepth ,, nWellSized ,, nLayout ,,
            circuit @(Width & Depth & WellSized & Layout);
```

Scans in Haskell Gibbons and Wu [2014] implement Scans as a shallow EDSL in Haskell (cf. Section 3.2 and Section 3.3). Variants of Scans are modeled as functions, thus adding new variants is simple through defining new functions. However, multiple (possibly dependent) operations are defined using tuples. Such an implementation is not modular because whenever a new operation is needed, existing code has to be modified for accommodating new operations. A follow-up Haskell implementation (cf. Section 3.4) employs a technique commonly known as Finally Tagless Carette et al. [2009], together with a type-class based encoding of subtyping on tuples, to modularize operation extensions. In essence, such an approach simulates subtyping and inheritance for enabling modular composition of the operations. However, this comes at the cost of boilerplate code and extra complexity due to additional parametrization that is needed to make the encoding work. Moreover, explicit delegations are required for defining dependent operations in the Haskell approach, making the code cumbersome to write. In contrast, CP avoids those issues by having built-in language support for nested composition, and compositional interfaces/traits and method patterns make the code quite easy to write.

Scans in Scala Chapter 3 presents a modular solution in Scala by combining the extensible Interpreter pattern [Wang and Oliveira, 2016] and Object Algebras [Oliveira and Cook, 2012]. Scans is modeled by a hierarchy of traits, where the root is an interface describing the operations a circuit supports while other traits concretely implement that interface. Adding new operations is done by defining another trait hierarchy that implements the extended interface and inherits existing traits. Through covariantly refining the circuit fields to the type of the extended interface, previously defined operations can be used as dependencies. Although such an implementation is modular, the extensions are linearly added, and the dependencies are strong due to the use of inheritance to express dependencies. Alternatively, the new trait hierarchy can be separately defined without inheriting existing ones. This weakens the dependencies but requires some boilerplate for gluing the hierarchies using mixin composition. Unlike CP where the composition is done *once* in the family level, *every* trait in Scala needs to be composed because Scala lacks support for nested composition. Another drawback is that Scala's constructors are *not virtual*. Directly calling **new** on constructors for creating objects results in non-modular code. Therefore, Object Algebras [Oliveira and Cook, 2012] are used for abstracting over the constructor calls, resulting in more boilerplate.

Scans in F_i^+ Bi et al. [2019] modularize Scans directly in F_i^+ using extensible records. Note that, because Scans do not have self-reference dependencies, there is no strict need for using traits, which are not directly supported in F_i^+. The syntax of Scans is defined similarly to `CircuitSig` except that `Circuit` is captured as a type parameter rather than a sort. The operation extensibility is achieved by defining new record instances while the variant extensibility is achieved by the intersection types and the merge operator. A key difference is how dependent operations are defined:

```
wellSized = {
  identity (n : Int) = { wS = true },
  fan      (n : Int) = { wS = true },
  above (c1 : WellSized&Width) (c2 : WellSized&Width) =
    { wS = c1.wS && c2.wS && c1.width == c2.width },
  beside (c1 : WellSized) (c2 : WellSized) = { wS  = c1.wS && c2.wS },
  stretch (ns : List Int) (c : WellSized&Width) =
    { wS = c.wS && length ns == c.width }
};
```

Note that `wellSized` is not given a type. Instead, it defines a record with various fields modeling constructors, and all the "constructor" arguments are explicitly annotated. This illustrates a crucial difference to the CP solution: while in CP `wellSized` (and other operations) implement a proper interface (via **implements**), in the F_i^+ encoding that is not the case. The dependency on `width` is loosely expressed by refining the circuit type as `WellSized&Width`. Such dependency is repeated several times in `above` and `stretch`. The lack of a proper interface when implementing operations allows for a relatively compact solution in F_i^+, but it has some important disadvantages. By implementing an interface in CP, we can ensure various things at the point of the operation definition. For instance, CP checks that: we implement all constructors, and all the constructors are defined with the right number of parameters and types for the parameters. In F_i^+ such checks are not done when defining operations, which is very error-prone. Errors like forgetting to implement a constructor or implementing a constructor with the wrong number of parameters or the wrong parameter types cannot be checked at the definition point, but are delayed to the composition point.

Evaluation Besides a qualitative analysis of the aforementioned modular implementations, we further evaluate them in terms of source lines of code (SLOC). To make the comparison fair, we have adapted their implementations to ensure that all the implementations are written in a similar programming style and provide the same functionalities. The SLOC for the modular implementations in Haskell, Scala, F_i^+ and CP are 87, 129, 72 and 70 respectively. CP's solution is the most compact one among the four implementa-

Table 5.1: *Different kinds of dependencies used in the mini interpreter.*

	Operation			
Dependency	eval	print	print(aux)	log
Child dependencies		✓		
Self dependencies		✓		✓
Mutual dependencies			✓	
Inherited attributes	✓			

tions, while also being the most modular one.

5.5.2 Mini Interpreter

Overview The second case study is a mini interpreter for an expression language. This case study is larger (around 700 SLOC) and more comprehensive than the previous one. Besides the EP and simple dependencies, it covers more forms of dependencies. In particular, self dependencies occur in this case study. Furthermore, the case study contains several uses of S-attributed grammars, polymorphic contexts, as well as multiple sorts.

The expression language consists of various sublanguages, including numeric and boolean literals, arithmetic expressions, logical expressions, comparisons, if-then-else branches, variable bindings, function closures, and function applications. The supported operations include a few variants of evaluation, pretty-printing, and logging. The sublanguages are separately defined as features [Prehofer, 1997], using different compositional interfaces and trait families. Through nested composition, these features can be arbitrarily combined to form a *product line of interpreters* [Pohl et al., 2005].

Dependencies The operations on the expression language contain non-trivial dependencies. Table 5.1 summarizes the different kinds of dependencies used in the mini interpreter. With techniques shown in Section 5.2.2, these dependencies are expressed in a modular way. For example, `log` is a simple form of logging, which shows the printing and evaluation results of an expression for debugging purposes. Here is a simplified logging implementation for numeric expressions:

```
logNum = trait implements NumSig<Eval&Print % Log> => {
  (Add e1 e2 [self:Eval&Print]).log = self.print ++ " is " ++ self.eval.toString;
  -- other constructors are omitted
};
```

To express self dependencies on `eval` and `print`, we annotate `Add`'s self-type as `Eval&Print`.

Polymorphic contexts There are four different kinds of contexts for evaluation in total: an empty context (`Top`); a map from strings to numbers for evaluating variable bindings; a map for dynamic scoping; an environment for intrinsic functions. Using techniques shown in Section 5.3.2, we model these contexts as polymorphic contexts to make the code with contexts modular and evolvable.

Multi-sorted languages Examples presented in the chapter so far are all based on a single-sorted expression language. Essentially, CP allows compositional interfaces to be parameterized by multiple sorts. This ability is demonstrated by the following code excerpt extracted from this case study:

```
type CmpSig<Boolean,Numeric> = {
  Eq  : Numeric -> Numeric -> Boolean;
  Cmp : Numeric -> Numeric -> Numeric;
  -- other constructors are omitted
};
```

`CmpSig` models equality and three-way comparison (also known as the *spaceship* operator), which returns 0 if the two operands are equal, 1 if the left operand is greater, or -1 otherwise. They take `Numeric` arguments and construct `Boolean` and `Numeric` traits respectively. Notice that `CmpSig` is developed as an independent feature. It can later be combined with other independently developed features such as numeric expressions:

```
expCmp N B = trait [self : NumSig<N>&CmpSig<B,N>] => {
  cmp = new Cmp (new Add (new Lit 1) (new Lit 2)) (new Lit 3);
};
```

In `cmp`, we construct an expression with the new constructor `Cmp`, as well as `Lit` and `Add` which are independently developed before.

5.5.3 C0 Compiler

Overview The C0 compiler was originally an educational one-pass compiler developed for the compilation course at Aarhus University. Rendel et al. [2014] translated this compiler into their encoding of Object Algebras, whereas we will present this case study in our approach of Compositional Programming. Then we will make a comparison with the implementation by Rendel et al. [2014], as well as the original non-modular version.

C0 selected a subset of the C programming language, consisting of only integer types, arithmetic, bitwise, and comparison operations, a few control flow statements, functions, and basic I/O. In other words, it is also a multi-sorted language, whose interfaces are parameterized by `Program`, `Function`, `Statement`, `Expression`, etc. A C0 program consists of function declarations and definitions, which will be compiled into Java bytecode. The original implementation was written in Java and later reimplemented in Scala using Object Algebras by Rendel et al. [2014]. Both implementations include a bytecode generator from AST nodes to JVM instructions as well as a recursive descent parser. Since the CP language is currently a research prototype and does not support I/O or complex string manipulation, we eliminate the parsing phase from this case study. Lexical analysis is not the core part of C0 and will not affect the validity of our evaluation.

Chained attributes We have shown various forms of dependencies in terms of *S-attributed grammars* in Section 5.2.2 and introduced polymorphic contexts to tackle *L-attributed grammars* in Section 5.3.3. However, there are still other kinds of dependencies in attribute grammars. In the C0 compiler, an attribute can depend on both its children and its parent or siblings. For example, consider an attribute that counts the number of leaf nodes (terminal symbols) occurring to the left or in the subtree of the current node. The attribute equations together with the production rules of `Lit` and `Add` are listed below:

$$E \to n \quad \texttt{\{Lit\}}$$

$$E.\texttt{count}_s = E.\texttt{count}_i + 1 \qquad (5.1)$$

$$E_0 \to E_1 \texttt{"+"} E_2 \quad \texttt{\{Add\}}$$

$$E_1.\texttt{count}_i = E_0.\texttt{count}_i \qquad (5.2)$$

$$E_2.\texttt{count}_i = E_1.\texttt{count}_s \qquad (5.3)$$

$$E_0.\texttt{count}_s = E_2.\texttt{count}_s \qquad (5.4)$$

Note that `count` has two roles: the subscript i stands for *inherited* attributes while s stands for *synthesized* ones.

For terminal symbols, we just add one to the inherited number, as Equation 5.1 shows. For nonterminal symbols, there are three attribute evaluation rules: Equation 5.2 is a trivial inherited attribute depending on its parent; Equation 5.3 is more interesting because it depends on its left sibling; Equation 5.4 is a trivial synthesized attribute depending on its child. These three equations compose a traversal of the syntax tree: $E_1.\texttt{count}_i$ inherits from its parent $E_0.\texttt{count}_i$ and then does its own computation on the left subtree to obtain $E_1.\texttt{count}_s$; $E_2.\texttt{count}_i$ inherits from its left sibling and then traverses the right subtree to obtain $E_2.\texttt{count}_s$; finally $E_0.\texttt{count}_s$ synthesizes the attribute from its child.

Such a tree traversal reveals an interesting class of attributes called *chained attributes* [Kastens and Waite, 1994]. A chained attribute is both inherited and synthesized. If we regard inherited attributes as *Reader* Monads and synthesized as *Writer*, then chained attributes correspond to *State* Monads.

In this case study, `HasVariables` and `HasFunctions` are chained attributes. They are used to do bookkeeping for variable and function declarations. Like inherited attributes, we model them with polymorphic contexts, where the inherited part serves as the context of the corresponding synthesized part:

```
type HasVariables = { variables : Map };                           -- inherited
type ChainedVariables Ctx = { variables : HasVariables&Ctx -> Map };  -- synthesized
```

If the chained attribute depends on other attributes, the context can be easily extended as we described in Section 5.3.2. Here is an example of extending the context with `HasOffset`:

```
type ParamSig<Parameter> = { Param : String -> Parameter };
parameterVariables (Ctx * (HasVariables&HasOffset)) =
  trait implements ParamSig<ChainedVariables (HasOffset&Ctx)> => {
    (Param id).variables (inh : HasOffset&HasVariables&Ctx) (name : String) =
      if name == id
      then inh.offset
      else inh.variables name;
  };
```

The constructor `Param` is used to declare a parameter within a function, and the trait implements the chained attribute of the variable environment. `(Param id).variables` will update the previous environment `inh.variables` with a new mapping from the given identifier to the current offset which works as the variable index. Note that, although these two `variables` have the same name, the former is a chained attribute while the latter is

Table 5.2: *Source lines of code for the three implementations of the C0 compiler.*

Java (Aarhus University)	SLOC	**Scala** (Rendel et al. [2014])	SLOC	**CP**	SLOC
Entangled Compiler	235	Generic	140	Maybe Algebra	12
(Tokenizer excluded)		Trees, Signatures and Combinators	558	Compositional Interfaces	32
		Composition and Assembly	101		
		Attribute Interfaces	32	Attribute Interfaces	8
		Algebra Implementations	191	Trait Implementations	216
Bytecode (Reformatted)	25	Bytecode Prelude	25	Bytecode Prelude	25
Main	14	Main	5	Main Example	21
Total	274	**Total**	1,052	**Total**	314

an inherited attribute. Such a delegation forms a tree traversal to do bookkeeping for parameter declarations.

Comparision The code statistics of the aforementioned three C0 implementations are shown in Table 5.2. The original Java implementation inlines semantic actions into the handwritten parser. For the sake of fairness, lexical analysis related lines are not counted and bytecode prelude are reformatted in the same style as the other two. Although the original code is slightly shorter than CP, it is highly entangled and hinders modularity and extensibility.

To modularize the original C0 implementation, Rendel et al. [2014] use an extended form of generalized Object Algebras to model attribute grammars in Scala. It allows L-attributed grammars with different kinds of dependencies to be modularly defined. Due to the lack of the proper composition mechanisms in Scala, the attributes cannot be easily composed, and specialized composition operators have to be explicitly defined. Such boilerplate code largely accounts for the SLOC reported in “Trees, Signatures and Combinators”. In addition, “Composition and Assembly” is the handwritten code to deal with various dependencies. Their “Attribute Interfaces” and “Algebra Implementations” are counterparts of our “Attribute Interfaces” and “Trait Implementations”.

Compared to CP, Rendel et al. [2014]’s approach is significantly more verbose (about 3.5x SLOC). In CP, we do not need to write boilerplate code for trait composition and only a few lines of “Compositional Interfaces” are necessary. A workaround they propose to avoid such boilerplate code is to employ meta-programming for generating the specialized signatures and combinators. However, their source code will not be type-checked before macro expansion. Compilation errors will be reported in terms of generated code, which could be confusing for programmers and make it hard to debug. Nevertheless, their assembly mechanism, which relies on specialized combinators that have to be handwritten, encodes the pattern of chained attributes and simplifies the algebra implementations. Our approach of polymorphic contexts is a little more complicated than theirs, but does not require any specialized combinators and works smoothly with nested trait composition.

5.6 Conclusion

We have presented key concepts of Compositional Programming together with a language design and implementation called CP. CP’s support for compositional interfaces, compositional traits, virtual constructors, and method patterns enables a programming style that allows programs with non-trivial dependencies to be modularized. The applicability of CP is demonstrated by various examples and case studies. The calculus that captures the essence of CP is proved to be type-safe.

We envision Compositional Programming as an alternative programming style to what is currently offered in both functional programming and object-oriented programming. Compositional Programming borrows many ideas from both paradigms. The style of Compositional Programming presented in this chapter is essentially a purely functional programming style, and draws inspiration from languages like Haskell. A purely functional style has benefits in terms of reasoning about programs and it also simplifies some issues related to the composition of code. Multiple inheritance in the presence of mutable state is a notorious problem, for instance. On the other hand, Compositional Programming also borrows ideas from object-oriented programming, namely by employing subtyping and nested composition (which is closely related to inheritance). This mix of ideas, together with some new ideas, results in a language that supports highly modular programs in a natural way.

Chapter 6

Related Work

There is a large body of literature on modular extensibility. This chapter only covers the closely related work.

6.1 Design Patterns for Modular Extensibility

This section discusses work on design patterns for solving the EP, which relates to all the thesis.

Polymorphic variants OCaml supports polymorphic variants [Garrigue, 1998]. Unlike traditional variants, polymorphic variant constructors are defined individually and are not tied to a particular datatype. Garrigue [Garrigue, 2000] presents a solution to the Expression Problem using polymorphic variants. To correctly deal with recursive calls, open recursion and an explicit fixed-point operator must be used properly. Otherwise, the recursion may go to the original function rather than the extended one. This causes additional work for the programmer, especially when the operation has complex dependencies. In contrast, Castor handles open recursion easily through OO dynamic dispatching, reducing the burden of programmers significantly.

Data types à la carte (DTC) DTC [Swierstra, 2008] encodes composable datatypes using existing features of Haskell. The idea is to express extensible datatypes as a fixpoint of co-products of functors. While it is possible to define operations that have dependencies or require nested pattern matching with DTC, the encoding becomes complicated and needs significant machinery. There is some follow-up work that tries to equip DTC with additional power. Bahr and Hvitved [2011] extend DTC with GADTs [Xi et al., 2003] and automatically generates boilerplate using Template Haskell [Sheard and Jones, 2002]. Oliveira et al. [2015] use list-of-functors instead of co-products to better simulate OOP features including subtyping, inheritance, and overriding.

Algebraic signatures and Object Algebras Readers familiar with algebraic signatures [Guttag and Horning, 1978] used in algebraic specification languages may observe some similarities to compositional interfaces. Algebraic signatures also allow the definition of constructors. However, the semantics of constructors in compositional interfaces differs from those in conventional algebraic signatures. The key difference is that the positive and negative occurrences of sorts are distinguished in CP, which is important to support an advanced form of modular dependencies but not well-supported with algebraic signatures.

Oliveira and Cook [2012] explored an encoding of algebraic signatures in OOP languages, yielding a solution to the EP called *Object Algebras*. Other design patterns, such as *Polymorphic Embeddings* [Hofer et al., 2008] and *Finally Tagless* [Carette et al., 2009] use similar encodings. Such encodings originate from the work by Hinze [2006] and Oliveira et al. [2006b]. Hinze [2006] showed how to model Church encodings of GADTs [Cheney and Hinze, 2002] using Haskell type classes. Oliveira et al. [2006b] then showed that such type class based encoding can also solve the EP. Object Algebras use parametric interfaces and classes to represent and implement the algebraic signatures, respectively. As discussed in Section 2.4.1, Object Algebras, in their basic form, are hard to be composed and express dependencies. These limitations are later addressed in Scala by means of *generalized Object Algebras* and intersection types, together with specialized combinators [Oliveira et al., 2013; Rendel et al., 2014]. However, the Scala approaches based on reflection or meta-programming have important drawbacks. In contrast, CP has built-in language support for sorts and nested composition, which is more convenient to use and avoids the use of reflection or meta-programming techniques.

6.2 Modular Pattern Matching

This section discusses previous work on modular forms of pattern matching, which relates to open pattern matching in Chapter 4 and method patterns in Chapter 5.

Open datatypes and open functions To solve the Expression Problem, Löh and Hinze [2006] propose to extend Haskell with open datatypes and open functions. Different from classic closed datatypes and closed functions, the open counterparts decentralize the definition of datatypes and functions and there is a mechanism that reassembles the pieces into a complete definition. To avoid unanticipated captures caused by classic *first-fit* pattern matching, a *best-fit* scheme is proposed, which rearranges patterns according to their specificness rather than the order (e.g. wildcards are least specific). However open datatypes and open functions are not supported in standard Haskell and more importantly, they do not support separate compilation: all source files of variants belonging to the same datatype must be available for code generation.

Object-Oriented pattern matching There are many attempts to bring notions similar to pattern matching into OOP. Multimethods [Chambers, 1992; Clifton et al., 2000] allow a series of methods of the same signature to co-exist. The dispatching for these methods additionally takes the runtime type of arguments into consideration so that the most specific method is selected. Pattern matching on multiple arguments can be simulated with multimethods. However, it is unclear how to do deep patterns with multimethods. Also, multimethods significantly complicate the type system. As we have discussed in

Section 4.2, case classes in Scala [Odersky et al., 2004] provide an interesting blend between algebraic datatypes and class hierarchies. Sealed case classes are very much like classical algebraic datatypes, and facilitate exhaustiveness checking at the cost of a closed (non-extensible) set of variants. Open case classes support pattern matching for class hierarchies, which can modularly add new variants. However no exhaustiveness checking is possible for open case classes. Besides case classes, extractors [Emir et al., 2007] are another alternative pattern matching mechanism in Scala. An extractor is a companion **`object`** with a user-defined `unapply` method that specifies how to tear down that object. Unlike case classes whose `unapply` method is automated and hidden, extractors are flexible, independent of classes but verbose. There are also proposals to extend mainstream languages with pattern matching such as Java. JMatch [Liu and Myers, 2003] extends Java with pattern matching using modal abstraction. JMatch methods additionally have backward modes that can compute the arguments from a given result, serving as patterns. Follow-up work [Isradisaikul and Myers, 2013] extends JMatch with exhaustiveness and totality checking on patterns in the presence of subtyping and inheritance. However, it requires a non-trivial language design with the help of an SMT solver. More recent OO languages like Newspeak [Geller et al., 2010] and Grace [Homer et al., 2012] are designed with first-class pattern matching, where patterns are objects and can easily be combined. To the best of our knowledge, none of these approaches fully meet the desirable properties summarized in Section 4.2.1.

6.3 Language Designs for Modular Extensibility

This section discusses features designed for modular extensibility, which relates to Chapter 4 and Chapter 5.

Mainstream statically typed OOP and virtual methods *Virtual methods* in OOP provide a way to weaken *method dependencies*. In a virtual method call, such as **`this`**`.m()` in a method of a class `A`, the call *does not* necessarily refer to the implementation of `m()` in `A`. Instead, it may refer to a later implementation in a subclass of `A`. The choice of the implementation of **`this`**`.m()` depends on which subclass of `A` is used to instantiate the object on which **`this`**`.m()` is called. However, virtual methods alone are insufficient to weaken other kinds of dependencies. Most programming languages tend to have *static* references to both *constructors* and *types*. For instance, if we refer to a constructor in Java, say **`new`** `A()`, then the constructor (unlike the method **`this`**`.m()`) *will always refer to the same constructor of class* `A`. Such *static* dependencies create a tight coupling between the use of the constructor and the class `A`, and make programs less modular than they ought to be. Moreover, most statically typed OOP languages use *(static) inheritance* pervasively. Inheritance often creates more coupling than needed between method implementations in subclasses and method implementations in the superclass. In a subclass declaration, such as **`class`** `A` **`extends`** `B {...}`, `B` must be some concrete class, with (possibly) some concrete method implementations. In other words, inheritance cannot be parametrized, and we cannot program against the *interface* of the superclass: we must program against some concrete class implementation.

CP adopts virtual methods, while also supporting virtual constructors to avoid static references to constructors. Static references to types, which would typically arise from constructors, are avoided by using sorts. Moreover, in CP most uses of inheritance can be replaced by code with weaker dependencies (see discussion in Section 5.2.2), thus avoiding the coupling problems introduced by inheritance. Altogether this leads to code that is more modular and has weaker dependencies than in conventional statically typed OOP languages.

Mixins and traits Single inheritance supported by many class-based OOP languages is insufficient for code reuse. On the other hand, multiple inheritance is hard to do correctly due to the existence of the *diamond problem*. Mixins [Bracha and Cook, 1990] provide one form of multiple inheritance. The Jigsaw framework [Bracha, 1992] formalizes mixins and provides a set of operators on mixins. There are other formalizations of mixins proposed for different languages such as ML-like languages [Ancona and Zucca, 2002; Duggan and Sourelis, 1996] and Java-like languages [Flatt et al., 1998; Lagorio et al., 2009]. Traits [Schärli et al., 2003] are an alternative to mixins. The fundamental difference between traits and mixins is the way of dealing with conflicts when composing multiple traits/mixins. Mixins *implicitly* resolve the conflicts according to the composition order whereas the programmer must *explicitly* resolve the conflicts for traits. The trait model avoids unexpected errors caused by the wrong choice of implementation through implicit resolution. Furthermore, it makes trait composition associative and commutative, and the order of traits being composed does not affect semantics (all permutations have the same behavior). This is in contrast with mixins, where composition is order-sensitive. Typically classes, mixins and traits in statically typed languages (such as Scala) are second-class and do not support dynamic inheritance. Dynamic languages like JavaScript can encode quite general forms of mixins and support dynamic inheritance. However, type-checking dynamic inheritance is hard. There is little work on typing first-class classes/mixins/traits. Takikawa et al. [2012]'s first-class classes in Typed Racket, Lee et al. [2015]'s tagged objects and Bi and Oliveira [2018]'s first-class traits are three notable works, which support such features in statically typed languages. Our work follows the first-class traits model by Bi and Oliveira [2018] and traits in CP are first-class, statically typed and support dynamic inheritance.

First-class traits and disjoint intersection types The theoretical foundation of CP is built on calculi supporting *disjoint intersection types* [Oliveira et al., 2016]. In particular, the semantics of CP is given by an elaboration to F_i^+ [Bi et al., 2019]. The F_i^+ calculus is the most advanced calculus in the line of work of disjoint intersection types. The λ_i calculus [Oliveira et al., 2016] was the first calculus with disjoint intersection types, and addressed the incoherence problem of intersection types with a merge operator [Dunfield, 2014] by introducing the notion of disjointness. The F_i calculus [Alpuim et al., 2017] extends λ_i with *disjoint polymorphism*. λ_i^+ [Bi et al., 2018] extends λ_i with BCD-style distributive subtyping, enabling nested composition. The F_i^+ calculus [Bi et al., 2019] combines disjoint intersection types, disjoint polymorphism, and nested composition, enabling all the foundational ingredients for Compositional Programming. However F_i^+ is still a core calculus, and has no support for *compositional interfaces*, *compositional traits*, *method patterns* and direct support for *nested trait composition*. These mechanisms, together with the mechanisms that enable weak dependencies, are all novel to the language design of CP.

Built upon F_i, SEDEL [Bi and Oliveira, 2018] is a statically-typed language that supports first-class traits and dynamic inheritance. The design of CP extends the design of SEDEL, and in particular it extends the design of *first-class traits* of SEDEL. However SEDEL does not support Compositional Programming. In addition to the new features specially designed for Compositional Programming, CP further takes the advantage of the unrestricted intersections brought by F_i^+ in improving the trait model proposed by SEDEL. In particular, the design of CP allows simpler typing rules compared to SEDEL. For example the typing rule for defining traits in SEDEL is:

$$\frac{\begin{array}{c}\overline{\Gamma, \textbf{self} : B \vdash E_i \Rightarrow \textbf{Trait}[B_i, C_i] \rightsquigarrow e_i}^{i \in 1..n} \\ \Gamma, \textbf{self} : B, \textbf{super} : C_1 \,\&\, .. \,\&\, C_n \vdash \{\overline{l_j = E'_j}^{j \in 1..m}\} \Rightarrow C \rightsquigarrow e \\ \overline{B <: B_i}^{i \in 1..n} \qquad \Gamma \vdash C_1 \,\&\, .. \,\&\, C_n \,\&\, C \qquad C_1 \,\&\, .. \,\&\, C_n \,\&\, C <: A\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{trait}[\textbf{self} : B]\ \textbf{inherits}\ \overline{E_i}^{i \in 1..n} \{\overline{l_j = E'_j}^{j \in 1..m}\} : A \Rightarrow \textbf{Trait}[B, A] \rightsquigarrow \\ \lambda(\textbf{self} : |B|).\ \textbf{let super} = \overline{(e_i\ \textbf{self})}^{i \in 1..n}\ \textbf{in super}\ ,,\ e\end{array}}$$

This rule, among others, is clearly more complicated than its counterpart (T-TRAIT) in CP. The complexity is mainly caused by dealing with expression sequences: every expression needs to be translated and validated. In contrast, CP processes only one expression thanks to the newly introduced rule MERGETRAIT. MERGETRAIT checks the disjointness of two traits and infers their merge as a trait type rather than an intersection type, thus reducing the complexity and duplication. Another important benefit of the CP design is improved support for *type inference*. In SEDEL, a type must be provided to instantiate a trait, but this type is inferred in CP. Moreover, parameters of a method pattern inside a trait can omit types in CP if they are declared by the type specified in the **implements** clause. This is quite handy for defining trait families.

Virtual classes and family polymorphism Ideas such as *virtual classes* [Madsen and Moller-Pedersen, 1989; Ernst et al., 2006] and *family polymorphism* [Ernst, 2001], extend the idea of virtual methods to classes. Thus, *classes* and *constructors* can themselves be *virtual*, weakening the dependencies to classes and constructors. Virtual classes were first introduced in the Beta language [Madsen et al., 1993]. Beta supports only single, static inheritance. The composition of Beta programs is done through the fragment system [Knudsen et al., 1994]. The gbeta language [Ernst, 1999] extends Beta with mixins and, more importantly, supports family polymorphism [Ernst, 2001]. Family polymorphism is a powerful mechanism for extensible and composable software design, which can solve the EP [Ernst, 2004]. Clarke et al. [2007] classify family polymorphism approaches into object family approaches and class family approaches. In an object family approach, nested classes are attributes of objects of the family class. Some examples of object family approaches are Beta, gbeta, CaesarJ [Aracic et al., 2006] and Newspeak [Bracha et al., 2010]. Whereas in a class family approach, nested classes are attributes of the family class. Class family approaches include Concord [Jolly et al., 2004], .FJ [Igarashi et al., 2005], Jx [Nystrom et al., 2004], J& [Nystrom et al., 2006] and Familia [Zhang and Myers, 2017]. There are also hybrid approaches like Tribe [Clarke et al., 2007]. Object family approaches are typically more expressive but require a more complex dependent type system, e.g. *vc* [Ernst et al., 2006]. The closest approach is Familia [Zhang and Myers, 2017], which also supports subtype polymorphism, family polymorphism, and parametric polymorphism but does not support the merge operator.

One difference between CP and the family polymorphism systems is that in those systems, inheritance is still used as a primary mechanism to express dependencies. Similarly to (regular) classes, the use of inheritance in family polymorphism sometimes creates more coupling than necessary between sub- and super-classes/families. In contrast, such dependencies can be weakened via CP's support for self-references and compositional interface type refinement, leading to more modular programs. Another difference is that conflicts are often implicitly resolved based on some order in those systems (e.g., gbeta uses the composition order and Jx uses the dispatch order). In contrast, CP adopts an approach where conflicts are explicitly resolved, following the trait model [Schärli et al., 2003].

ML modules and Scala The design of CP is partly inspired by ML module systems [MacQueen, 1984]. Analogously to compositional interfaces and trait families in CP, ML signatures and structures can be used to specify and implement the constructors. However, unlike CP, ML modules are neither extensible nor first-class. There are many proposals to extend the ML modules with additional expressiveness, such as making modules first-class [Russo, 2000] or recursive [Russo, 2001]. Together with other features, ML modules can also be used in solving the EP [Nakata and Garrigue, 2006].

CP is also influenced by Scala [Odersky et al., 2004], where features such as intersection types, traits, and self-types are shared. Scala's traits are not first-class and do not support dynamic inheritance and nested composition. Nevertheless, Scala supports virtual types [Madsen and Moller-Pedersen, 1989; Igarashi and Pierce, 1999], which can be used for simulating family polymorphism but in a much more verbose way [Odersky and Zenger, 2005]. Sorts in CP are modeled in a way similar to type parameters. Another option is to use virtual types. The strengths and weaknesses of type parameters and virtual types are summarized by Bruce et al. [1998]: type parameters are flexible in composing and decomposing types while virtual types are good at specifying mutually recursive families of classes whose relationships are preserved in the derived family. Type parameters are a natural choice for CP since the underlying F_i^+ calculus supports disjoint polymorphism. In future work, we would like to explore a design with virtual types.

6.4 Metaprogramming for Modular Extensibility

This section discusses approaches that use metaprogramming to improve modularity, which relates to Chapter 4 and Chapter 5.

SOP, MDSoC, AOP, and FOP *Subject-oriented programming* (SOP) [Harrison and Ossher, 1993], *multi-dimensional separation of concerns* (MDSoC) [Tarr et al., 1999], *aspect-oriented programming* (AOP) [Kiczales et al., 1997], and *feature-oriented programming* (FOP) [Prehofer, 1997] are software paradigms that share a similar vision of *separation of concerns*: i.e., separating a program into different parts so that each part addresses a separate concern. Since in those paradigms, a complete program has been separated, a *composition* mechanism to assemble the parts back together is necessary. Typically SOP, MDSoC, AOP and FOP employ a *meta-programming* approach to software composition. Such meta-programming approaches are usually an extension to an existing programming language, such as Hyper/J and AspectJ for Java. A source-to-source compiler will combine the separated aspects before producing plain Java code. Quite often many of those tools do not fully support modular type-checking or separate compilation.

In contrast, Compositional Programming is a language-based approach, with both a clearly defined static and dynamic semantics. The merge operator provides the composition mechanism in Compositional Programming. What distinguishes the elaboration adopted by CP from general meta-programming is that the elaboration is completely transparent for a programmer: 1) Type-checking is done directly in the source language, where type errors (and other well-formedness errors) in programs are reported in terms of the source rather than the target; 2) Type-checking is modular: each definition can be type-checked with only its implementation and type signatures of the dependencies. Worth noting is that the style of elaboration employed by CP to give the semantics to the language is also adopted by other languages. Most notably the GHC Haskell compiler elaborates the source language (Haskell) into a small core language [Sulzmann et al., 2007]. Like CP, all type-checking is done at the source level and the elaboration process is transparent to Haskell programmers. In contrast, in many approaches that employ meta-programming, often there is no source-level type-checking or even some more basic error checking like syntax well-formedness. Consequently, no modular type-checking is offered and errors are reported on the generated program, which are hard to understand.

Language workbenches Language workbenches [Fowler, 2005; Erdweg et al., 2013] have been proposed for reducing the engineering effort involved in software language development. Modularity is an important concern in language workbenches for allowing existing language components to be reused in developing new languages [Combemale et al., 2018]. Traditionally most of the work on language workbenches has focused on *syntatic modularity* approaches. More *semantic modularity* aspects such as separate compilation and modular typechecking are not well addressed. However, more recent work on language workbenches has started to incorporate semantic modularity techniques. We compare our work next, to the language workbenches that employ semantic modularization techniques. With Neverlang [Vacchi and Cazzola, 2015], users do not directly program with visitors. Instead, they have to use a DSL and learn specific concepts such as slice and roles. MontiCore [Heim et al., 2016] generates the visitor infrastructure from its grammar specification. To address the extensibility issue, MontiCore overrides the `accept` method and uses casts for choosing the right visitor for extended variants, thus is not type-safe. Also, MontiCore supports imperative style visitors only. Alex [Leduc et al., 2018] also provides a form of semantic modularity based on the *Revisitor* pattern [Leduc et al., 2017], which can be viewed as a combination of Object Algebras and Walkabout [Palsberg and Jay, 1998]. By moving the dispatching method from the class hierarchy to the visitor interface, the *Revisitor* pattern can work for legacy class hierarchies that do not anticipate the usage of visitors. However, the dispatching method generated by Alex is implemented using casts and has to be modified whenever new variants are added, thus is neither modular nor type-safe. Castor fully supports semantic modularity and allows users to do the development using their familiar language with a few annotations. For the moment, Castor still lacks much of the functionality for various other aspects of language implementations that are covered by language workbenches. Nevertheless, the modularization techniques employed by Castor could be useful in the context of language workbenches to improve reuse and type-safety of language components, in the same way that visitors are used in Neverlang and Revisitors are used in Alex.

Chapter 7

Future Work

The three approaches proposed in this thesis have their own strengths and weaknesses and complement each other. This chapter discusses the limitations of the proposed solutions and possible directions for future work.

7.1 Plain Design Patterns

In Chapter 3, we show how to modularize shallow EDSLs using a combination of two design patterns, Extensible Interpreters and Object Algebras. Such an approach has two major limitations. Firstly, it is problematic to model binary and producer operations using the Extensible Interpreter pattern since the AST type evolves in extensions. Although the signature of producer operations can be updated through covariant refinement, the implementations have to be duplicated. For binary operations, it is worse because it is not possible to refine the signature since AST types occur in contravariant positions. As a result, the new version of that binary operation must coexist with the old one, which is non-modular and error-prone. Secondly, the combination of Extensible Interpreters and Object Algebras brings more expressiveness and modularity but at the same time complicates the encoding. One direction of future work is to employ metaprogramming to eliminate some of the boilerplate, just like what we have done in Castor. Essentially, the trait hierarchies can be generated by analyzing the Object Algebra interface or vice versa. For example, from the following Object Algebra interface:

```
trait CircuitAlg[Circuit] {
  def Id(n: Int): Circuit
  def Fan(n: Int): Circuit
  def Beside(c1: Circuit, c2: Circuit): Circuit
  def Above(c1: Circuit, c2: Circuit): Circuit
  def Stretch(ns: List[Int], c: Circuit): Circuit
}
```

we can generate the hierarchy below:

```
trait Circuit {
}
trait Id extends Circuit {
  val n: Int
}
trait Fan extends Circuit {
  val n: Int
}
trait Besides extends Circuit {
  val c1, c2: Circuit
}
trait Above extends Circuit {
  val c1, c2: Circuit
}
trait Stretch extends Circuit {
  val ns: List[Int]; val c: Circuit
}
```

We can see a close connection between the two code snippets, where the type parameter of the Object Algebra interface becomes the root of the hierarchy and each factory method has a trait definition with arguments captured as fields. With the hierarchy generated, the programmer only needs to fill in the methods in each trait. Similarly, more code can be generated such as the factory for creating objects from the trait hierarchy. However, with metaprogramming, the benifits of a plain design pattern are lost. Therefore, another direction is to seek powerful yet simple design patterns for modularity.

7.2 Castor

While Castor is practical and serves the purpose of programming with visitors, there are important drawbacks on such a meta-programming, library-based approach:

- **Unnecessary annotations.** With the current version of Scalameta, we are not able to get information from annotated parents. If parents' information were accessible, the inherited datatypes and visitors could be analyzed and `@adts` and `@ops` annotations could be eliminated.

- **Boilerplate for nested composition.** Lacking of parents' information also disallows automatically composing nested members. Consequently, the composition has to be repeated for each member of the family, which is quite tedious.

- **Imprecise error messages.** As Castor modifies the annotated programs, what the compiler reports are errors on the modified program rather than the original program. Reasoning about the error messages becomes harder as they are mispositioned and require some understanding of the generated code.

One direction of future work is to further simplify programming with Castor, Assuming that there is sufficient support from the underlying metaprogramming library, the implementation of the Arith language can be simplified as:

```
@family trait Arith extends Nat with Bool {
  @adt trait Tm {
    case class TmIsZero(t: Tm)
  }
  @visit(Tm) trait Eval1 {
    def tmIsZero = {
      case TmIsZero(TmZero) => TmTrue
      case TmIsZero(TmSucc(t)) if nv(t) => TmFalse
      case TmIsZero(t) => TmIsZero(this(t))
    }
  }
}
```

where the extends clause is expressed only once at the top level and extends clauses for nested members such as super[Nat].Tm with super[Bool].Tm are inferred. Still, the syntax and semantics of Scala cannot be changed to enforce certain restrictions such as enforcing a default for nested patterns.

Another direction of future work is to integrate the idea of Castor into CP that additionally supports *open datatypes* and *open pattern matching*. Here, we sketch out how the code for implementing Arith would be like in the extended CP language:

```
type Term = {
  data Tm; -- Open datatype
  eval1 : Tm -> Tm;
};
type Nat extends Term = {
  TmZero : Tm;       -- Constructor for Tm
  TmSucc : Tm -> Tm; -- Constructor for Tm
  TmPred : Tm -> Tm; -- Constructor for Tm
  nv : Tm -> Boolean;
};
nat = trait implements Nat => {
  TmZero.nv = true;
  (TmSucc t).nv = t.nv;
  _.nv = false; -- Local wildcard

  (TmSucc t).eval1 = TmSucc t.eval1;
  (TmPred TmZero).eval1 = TmZero;
  (TmPred (TmSucc t)).eval1 if t.nv = t;
  (TmPred t).eval1 = TmPred t.eval1;
  _.eval1 = throw NoRuleApplies;
};
-- The code for bool omitted
type Arith extends Nat & Bool = {
  TmIsZero : Tm -> Tm;
};
arith = trait implements Arith inherits nat ,, bool => {
  (TmIsZero TmZero).eval1 = TmTrue;
  (TmIsZero (TmSucc t)).eval1 if t.nv = TmFalse;
  (TmIsZero t).eval1 = TmIsZero t.eval1;
};
```

Without the restriction of existing languages, the syntax and semantics can be much more flexible and coherent. Both top-level (shallow) and nested (deep) patterns are written using a unified syntax and patterns with distinct top-level constructors are by default order-insensitive. Moreover, we can enforce that nested patterns must come with a default case. As we shall discuss later, supporting such a design in CP is still challenging.

7.3 CP

As a prototype design for Compositional Programming, CP is still far from being a fully-fledged programming language for real-world software development. Some possible directions for future work include the addition of *recursive types* and *type constructors*, *mutable state* for modeling imperative objects, and improvements on *type-inference*.

In our view, the addition of recursive types is most pressing as there are many use cases for such signatures. For example, with recursive types, we can model binary methods [Bruce et al., 1995], or operations that return the same type that is being processed. For example, with recursive types, we should be able to model the double operation described by Zenger and Odersky [2005]. This operation doubles the literals in an arithmetic expression, where each constructor implements the following interface:

```
type Dbl = mu Exp. { double : Exp };
```

`Exp` is captured as a recursive type. On the other hand, supporting type constructors allows us to model, for example, the compositional interface of streams adapted from Biboudis et al. [2015]'s work:

```
type StreamSig<F : Type -> Type> = {
  Source : forall T. Array T -> F T;
  Map : forall T R. (T -> R) -> F T -> F R;
  FlatMap : forall T R. (T -> F R) -> F T -> F R;
  Filter : (T -> Bool) -> F T -> F T;
};
```

where the sort `F` is a type constructor (i.e. a function on types). Extending CP with recursive types and type constructors is non-trivial. The first step is to study how recursive types and type constructors interact with disjoint intersection types and other features of CP.

Although the programming style of CP is functional, a natural question is whether the ideas of Compositional Programming can be adapted into a programming model with imperative objects. There are several challenges here. One of them is to see how mutable state can be integrated into calculi with disjoint intersection types and a merge operator. A starting point in this direction is the work by Blaauwbroek [2017], which studies the addition of mutable references into calculus with intersection types and a merge operator. Another general challenge is that, once imperative objects are supported, we must face the issues of multiple inheritance in the presence of mutable state, which is a well-known source of problems.

Finally, CP has some support for type inference, such as inferring the constructor parameters from the **`implements`** clause. However, this support is rather limited. In particular, uses of polymorphic definitions must explicitly pass all type arguments. It will be interesting to investigate *local type inference* [Pierce and Turner, 2000] to infer some of those arguments and improve the convenience of using polymorphic definitions. A more ambitious direction would be looking into MLsub [Dolan and Mycroft, 2017] and see whether it would be possible to adapt or extend MLsub type inference to CP. The work by van den Berg [2020] is a starting point in this direction.

Chapter 8

Conclusion

Supporting modular extensible software development is a longstanding problem. In this thesis, we have presented different approaches to modular extensible software development from the programming languages perspective. The proposed solutions can solve not only canonical modularity problems such as the Expression Problem but also model programs with non-trivial dependencies. In particular, we have explored:

- *Plain design patterns*, which can be directly applied to existing programming languages. We have shown that with a combination of two design patterns, the Extensible Interpreter pattern and Object Algebras, shallow EDSLs can be made more modular.

- *Metaprogramming-based design patterns*, which further eliminate boilerplate incurred by design patterns. The proposed CASTOR framework employs annotations to allow us to program with extensible visitors without the pain of writing boilerplate code.

- *Novel language designs*, which allow specialized syntax and semantics designed for modularity. With such flexibility, we present CP, a language design for Compositional Programming.

Although there is still a gap between the current solutions and the ideal way of modular extensible software development, it is a meaningful step forward. We hope this thesis can give programmers and language designers some insights on how to write modular extensible code and how to support modular extensible code development from the programming language perspective.

Bibliography

João Alpuim, Bruno C d S Oliveira, and Zhiyuan Shi. 2017. Disjoint polymorphism. In *European Symposium on Programming*. Springer, 1–28.

Davide Ancona and Elena Zucca. 2002. A calculus of module systems. *Journal of functional programming* 12, 2 (2002), 91–132.

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. 2006. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*. Springer, 135–173.

Patrick Bahr and Tom Hvitved. 2011. Compositional Data Types. In *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming (WGP '11)*. 83–94. https://doi.org/10.1145/2036918.2036930

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A filter lambda model and the completeness of type assignment 1. *The journal of symbolic logic* 48, 4 (1983), 931–940.

Xuan Bi and Bruno C d S Oliveira. 2018. Typed first-class traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Xuan Bi, Bruno C d S Oliveira, and Tom Schrijvers. 2018. The essence of nested composition. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Xuan Bi, Ningning Xie, Bruno C d S Oliveira, and Tom Schrijvers. 2019. Distributive Disjoint Polymorphism for Compositional Programming. In *European Symposium on Programming*. Springer, 381–409.

Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. 2015. Streams a la carte: Extensible pipelines with object algebras. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lasse Blaauwbroek. 2017. *On the Interaction Between Unrestricted Union and Intersection Types and Computational Effects*. Master's thesis. Technical University Eindhoven.

Gilad Bracha. 1992. *The programming language jigsaw: mixins, modularity and multiple inheritance*. Ph.D. Dissertation. Dept. of Computer Science, University of Utah.

Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90)*.

Gilad Bracha, Peter Von Der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as objects in newspeak. In *European Conference on Object-Oriented Programming*. Springer, 405–428.

Kim Bruce, Luca Cardelli, Giuseppe Castagna, Hopkins Objects Group, Gary T Leavens, and Benjamin Pierce. 1995. On binary methods. *Theory and Practice of Object Systems* 1, 3 (1995), 221–242.

Kim Bruce, Martin Odersky, and Philip Wadler. 1998. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*.

Peter Buchlovsky and Hayo Thielecke. 2006. A Type-theoretic Reconstruction of the Visitor Pattern. *Electron. Notes Theor. Comput. Sci.* 155 (May 2006), 309–329. https://doi.org/10.1016/j.entcs.2005.11.061

Eugene Burmako. 2017. *Unification of Compile-Time and Runtime Metaprogramming in Scala*. Ph.D. Dissertation. EPFL.

R. M. Burstall, D. B. MacQueen, and D. T. Sannella. 1981. *HOPE: An experimental applicative Language*. Technical Report CSR-62-80. Computer Science Dept, Univ. of Edinburgh.

Luca Cardelli. 1994. *Extensible Records in a Pure Calculus of Subtyping*. MIT Press, Cambridge, MA, USA.

Luca Cardelli and John Mitchell. 1991. Operations on Records. *Mathematical Structures in Computer Science* 1 (1991), 3–48. Also in ; available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. *Journal of Functional Programming* 19, 05 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

Walter Cazzola and Edoardo Vacchi. 2016. Language Components for Modular DSLs Using Traits. *Computer Languages, Systems & Structures* 45 (2016), 16–34. https://doi.org/10.1016/j.cl.2015.12.001

Craig Chambers. 1992. Object-Oriented Multi-Methods in Cecil. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 92)*. https://doi.org/10.5555/646150.679216

James Cheney and Ralf Hinze. 2002. A lightweight implementation of generics and dynamics, Manuel M.T. Chakravarty (Ed.). ACM, New York, NY, USA, 90–104. https://doi.org/10.1145/581690.581698

Alonzo Church. 1936. An unsolvable problem of elementary number theory. *American journal of mathematics* 58, 2 (1936), 345–363.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *Proceedings of the 6th international conference on Aspect-oriented software development*. 121–134.

Curtis Clifton, Gary T Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *ACM Sigplan Notices*, Vol. 35. ACM, 130–145.

Benoit Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, et al. 2018. Concern-oriented language development (cold): Fostering reuse in language engineering. *Computer Languages, Systems & Structures* 54 (2018), 139–155.

William Cook and Jens Palsberg. 1989. A Denotational Semantics of Inheritance and Its Correctness. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 89)*. ACM, New York, NY, USA, 433443. https://doi.org/10.1145/74877.74922

William R. Cook. 2009. On Understanding Data Abstraction, Revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. 557–572. https://doi.org/10.1145/1639949.1640133

Stephen Dolan and Alan Mycroft. 2017. Polymorphism, Subtyping, and Type Inference in MLsub *(POPL 2017)*. ACM, New York, NY, USA, 60–72. https://doi.org/10.1145/3009837.3009882

Dominic Duggan and Constantinos Sourelis. 1996. Mixin modules. *ACM SIGPLAN Notices* 31, 6 (1996), 262–273.

Joshua Dunfield. 2014. Elaborating Intersection and Union Types. *Journal of Functional Programming* 24, 2-3 (2014), 133–165. https://doi.org/10.1017/S0956796813000270

Burak Emir, Martin Odersky, and John Williams. 2007. Matching objects with patterns. In *European Conference on Object-Oriented Programming*.

Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2013. The state of the art in language workbenches. In *International Conference on Software Language Engineering*.

Erik Ernst. 1999. *gbeta-a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph.D. Dissertation. University of Aarhus.

Erik Ernst. 2001. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*.

Erik Ernst. 2004. The expression problem, Scandinavian style. *On Mechanisms For Specialization* (2004), 27.

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A Virtual Class Calculus. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*.

Martin Erwig and Eric Walkingshaw. 2012. Semantics-Driven DSL Design. In *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. 56–80. https://doi.org/10.4018/978-1-4666-2092-6.ch003

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. 1998. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 171–183.

Martin Fowler. 2005. Language workbenches: The killer-app for domain specific languages. http://martinfowler.com/articles/languageWorkbench.html

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*.

Jacques Garrigue. 2000. Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering*.

Felix Geller, Robert Hirschfeld, and Gilad Bracha. 2010. *Pattern Matching for an object-oriented and dynamically typed programming language*. Number 36. Universitätsverlag Potsdam.

Jeremy Gibbons. 2003. Origami Programming. 41–60. http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/origami.pdf

Jeremy Gibbons and Nicolas Wu. 2014. Folding Domain-Specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. 339–347. https://doi.org/10.1145/2628136.2628138

John V. Guttag and James J. Horning. 1978. The algebraic specification of abstract data types. *Acta informatica* 10, 1 (1978), 27–52.

William Harrison and Harold Ossher. 1993. Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. 411–428.

Robert Heim, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Andreas Wortmann. 2016. Compositional Language Engineering Using Generated, Extensible, Static Type-Safe Visitors. In *European Conference on Modelling Foundations and Applications*.

George T. Heineman and William T. Councill (Eds.). 2001. Component-Based Software Engineering: Putting the Pieces Together. (2001).

Ralf Hinze. 2004. An Algebra of Scans. In *Mathematics of Program Construction*. 186–210. https://doi.org/10.1007/978-3-540-27764-4_11

Ralf Hinze. 2006. Generics for the Masses. *Journal of Functional Programming* 16, 4-5 (2006), 451–483. https://doi.org/10.1017/S0956796806006022

Christian Hofer and Klaus Ostermann. 2010. Modular Domain-specific Language Components in Scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*.

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of Dsls. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE '08)*.

Michael Homer, James Noble, Kim B. Bruce, Andrew P. Black, and David J. Pearce. 2012. Patterns As Objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. New York, NY, USA, 17–28.

Xuejing Huang and Bruno C. d. S. Oliveira. 2020. A Type-Directed Operational Semantics For a Calculus with a Merge Operator. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 166. https://doi.org/10.4230/LIPIcs.ECOOP.2020.26

Paul Hudak. 1998. Modular Domain Specific Languages and Tools. In *Proceedings. Fifth International Conference on Software Reuse*. 134–142. https://doi.org/10.1109/ICSR.1998.685738

Atsushi Igarashi and Benjamin C Pierce. 1999. Foundations for virtual types. In *European Conference on Object-Oriented Programming*. Springer, 161–185.

Atsushi Igarashi, Chieri Saito, and Mirko Viroli. 2005. Lightweight family polymorphism. In *Asian Symposium on Programming Languages and Systems*. Springer, 161–177.

Chinawat Isradisaikul and Andrew C. Myers. 2013. Reconciling Exhaustive Pattern Matching with Objects. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*.

Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. 2004. Simple dependent types: Concord. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*.

Simon Peyton Jones. 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Vojin Jovanovic, Amir Shaikhha, Sandro Stucki, Vladimir Nikolaev, Christoph Koch, and Martin Odersky. 2014. Yin-Yang: Concealing the Deep Embedding of DSLs. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. 73–82. https://doi.org/10.1145/2658761.2658771

Uwe Kastens and William M. Waite. 1994. Modularity and reusability in attribute grammars. *Acta Informatica* 31, 7 (1994), 601–627.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *European conference on object-oriented programming*. Springer, 220–242.

Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Generic and Indexed Programming*. 130–174. https://doi.org/10.1007/978-3-642-32202-0_3

Jorgen Lindskov Knudsen, Boris Magnusson, Mats Lofgren, and Ole L Madsen. 1994. *Object Oriented Software Development Environments: The Mjolner Approach*. Prentice-Hall, Inc.

Donald E. Knuth. 1968. Semantics of Context-Free Languages. *Math. Sys. Theory* 2, 2 (1968), 127–145.

Donald E. Knuth. 1990. The Genesis of Attribute Grammars. In *WAGA*. 1–12.

Giovanni Lagorio, Marco Servetto, and Elena Zucca. 2009. Featherweight jigsaw: A minimal core calculus for modular composition of classes. In *European Conference on Object-Oriented Programming*. Springer, 244–268.

Manuel Leduc, Thomas Degueule, and Benoit Combemale. 2018. Modular language composition for the masses. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*.

Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs Van Der Storm, and Olivier Barais. 2017. Revisiting visitors for modular extension of executable DSMLs. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*.

Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. 2015. A theory of tagged objects. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Jed Liu and Andrew C Myers. 2003. JMatch: Iterable abstract pattern matching for Java. In *PADL*.

Andres Löh and Ralf Hinze. 2006. Open data types and open functions. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*.

David MacQueen. 1984. Modules for standard ML. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 198–207.

Ole Lehrmann Madsen and Birger Moller-Pedersen. 1989. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications*. 397–406. https://doi.org/10.1145/74877.74919

Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1993. *Object-oriented programming in the BETA programming language*. Addison-Wesley.

Robert C. Martin. 2002. *The Principles, Patterns, and Practices of Agile Software Development*. Prentice Hall.

Tanja Mayerhofer and Manuel Wimmer. 2015. The TTC 2015 Model Execution Case.. In *TTC@ STAF*. 2–18.

Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. 706–706. https://doi.org/10.1145/1142473.1142552

Bertrand Meyer. 1988. *Object-Oriented Software Construction*. Prentice Hall.

Bertrand Meyer and Karine Arnout. 2006. Componentization: the Visitor example. *Computer* 39, 7 (2006), 23–30.

Todd Millstein, Colin Bleckner, and Craig Chambers. 2004. Modular Typechecking for Hierarchically Extensible Datatypes and Functions. *ACM Trans. Program. Lang. Syst.* 26, 5 (Sept. 2004).

Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. 1997. The Definition of Standard ML-Revised. (1997).

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Generics of a Higher Kind. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. https://doi.org/10.1145/1449764.1449798

Keiko Nakata and Jacques Garrigue. 2006. Recursive modules for programming. *ACM SIGPLAN Notices* 41, 9 (2006), 74–86.

Peter Naur. 1968. Software engineering-report on a conference sponsored by the NATO Science Committee Garimisch, Germany. *http://homepages. cs. ncl. ac. uk/brian. randell/NATO/nato1968. PDF* (1968).

Martin E Nordberg III. 1996. Variations on the visitor pattern. In *PLoP96 Writers Workshop*, Vol. 154.

Nathaniel Nystrom, Stephen Chong, and Andrew C Myers. 2004. Scalable extensibility via nested inheritance. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 99–115.

Nathaniel Nystrom, Xin Qi, and Andrew C Myers. 2006. J& nested intersection for scalable software composition. *ACM SIGPLAN Notices* 41, 10 (2006), 21–36.

Martin Odersky. 2006. Pimp My Library. http://www.artima.com/weblogs/viewpost.jsp?thread=179766 Last accessed on 2019-01-29.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. *An overview of the Scala programming language*. Technical Report.

Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005)*.

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components. In *Proceedings of the 23rd European Conference on Object-Oriented Programming*.

Bruno C. d. S. Oliveira and William R. Cook. 2012. Extensibility for the Masses: Practical Extensibility with Object Algebras. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP '12)*. https://doi.org/10.1007/978-3-642-31057-7_2

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006a. Extensible and modular generics for the masses. *Trends in Functional Programming* 7 (2006), 199–216.

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006b. Extensible and Modular Generics for the Masses. In *Trends in Functional Programming*. 199–216.

Bruno C. d. S. Oliveira, Shin-Cheng Mu, and Shu-Hung You. 2015. Modular Reifiable Matching: A List-of-functors Approach to Two-level Types. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*.

Bruno C d S Oliveira, Zhiyuan Shi, and João Alpuim. 2016. Disjoint intersection types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 364–377.

Bruno C. d. S. Oliveira, Tijs van der Storm, Alex Loh, and William R. Cook. 2013. Feature-Oriented Programming with Object Algebras. In *Proceedings of the 27th European Conference on Object-Oriented Programming*. https://doi.org/10.1007/978-3-642-39038-8_2

Jens Palsberg and C. Barry Jay. 1998. The Essence of the Visitor Pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*.

Tanumoy Pati and James H Hill. 2014. A survey report of enhancements to the visitor software design pattern. *Software: Practice and Experience* 44, 6 (2014), 699–733.

Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI '88)*. https://doi.org/10.1145/53990.54010

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

Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (Jan. 2000), 44.

Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media.

Erik Poll. 1997. System F with Width-Subtyping and Record Updating. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software (TACS 1997)*. Springer-Verlag, Berlin, Heidelberg.

Christian Prehofer. 1997. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*. Springer, 419–443.

Tillmann Rendel, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2014. From Object Algebras to Attribute Grammars. In *Proceedings of the 2014 ACM International Conference on Object-Oriented Programming Systems Languages and Applications*.

John C. Reynolds. 1978. *User-defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction*. 309–317. https://doi.org/10.1007/978-1-4612-6315-9_22

Tiark Rompf and Nada Amin. 2015. Functional Pearl: A SQL to C Compiler in 500 Lines of Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. 2–9. https://doi.org/10.1145/2784731.2784760

Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. 127–136. https://doi.org/10.1145/1868294.1868314

Claudio V Russo. 2000. First-class structures for Standard ML. In *European Symposium on Programming*. Springer, 336–350.

Claudio V Russo. 2001. Recursive structures for Standard ML. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*. 50–61.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P Black. 2003. Traits: Composable units of behaviour. In *European Conference on Object-Oriented Programming*. Springer, 248–274.

DS Scott. 1963. A system of functional abstraction. *Unpublished manuscript* (1963).

Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*.

Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton-Jones, and K. Donnelly. 2007. System F with type equality coercions. In *TLDI*.

Josef Svenningsson and Emil Axelsson. 2012. Combining Deep and Shallow Embedding for EDSL. In *Trends in Functional Programming*. 21–36. https://doi.org/10.1007/978-3-642-40447-4_2

Wouter Swierstra. 2008. Data Types à la Carte. *Journal of Functional Programming* 18, 04 (2008), 423–436. https://doi.org/10.1017/S0956796808006758

Asumu Takikawa, T Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual typing for first-class classes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 793–810.

Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*. IEEE, 107–119.

Kresten Krab Thorup. 1997. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*.

Mads Torgersen. 2004. The expression problem revisited. In *European Conference on Object-Oriented Programming*.

Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Computer Languages, Systems & Structures* 43 (2015), 1–40.

Birthe van den Berg. 2020. ICFP: G: Type Inference for Disjoint Intersection Types.

Joost Visser. 2001. Visitor Combination and Traversal Control. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. https://doi.org/10.1145/504282.504302

Philip Wadler. 1992. The essence of functional programming. *19th POPL* (Jan. 1992), 1–14.

Philip Wadler. 1998. The Expression Problem. (Nov. 1998). Note to Java Genericity mailing list.

Yanlin Wang and Bruno C. d. S. Oliveira. 2016. The Expression Problem, Trivially!. In *Proceedings of the 15th International Conference on Modularity (MODULARITY 2016)*. 37–41. https://doi.org/10.1145/2889443.2889448

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*.

Matthias Zenger and Martin Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*.

Mathhias Zenger and Martin Odersky. 2005. Independently Extensible Solutions to the Expression Problem. In *Foundations of Object-Oriented Languages (FOOL)*.

Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira, and Tijs van der Storm. 2015. Scrap Your Boilerplate with Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 127–146. https://doi.org/10.1145/2814270.2814279

Weixin Zhang. 2017. Extensible Domain-Specific Languages in Object-Oriented Programming. *HKU Theses Online (HKUTO)* (2017).

Weixin Zhang and Bruno C.d.S. Oliveira. 2020. Castor: Programming with extensible generative visitors. *Science of Computer Programming* 193 (2020), 102449. https://doi.org/10.1016/j.scico.2020.102449

Weixin Zhang and Bruno C. d. S. Oliveira. 2017. EVF: An Extensible and Expressive Visitor Framework for Programming Language Reuse. In *31st European Conference on Object-Oriented Programming*. https://doi.org/10.4230/LIPIcs.ECOOP.2017.29

Weixin Zhang and Bruno C. d. S. Oliveira. 2018. Pattern Matching in an Open World. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. https://doi.org/10.1145/3278122.3278124

Weixin Zhang and Bruno C. d. S. Oliveira. 2019. Shallow EDSLs and Object-Oriented Programming: Beyond Simple Compositionality. *The Art, Science, and Engineering of Programming* 3, 3 (2019), 1–25.

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021. Compositional Programming. *ACM Transactions on Programming Languages and Systems* (2021), to appear.

Yizhou Zhang and Andrew C Myers. 2017. Familia: unifying interfaces, type classes, and family polymorphism. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–31.

Appendices

Appendix A

Full Type System of CP

$\boxed{\Delta, \Sigma \vdash A \Rightarrow B}$

E-TOP
$$\Delta, \Sigma \vdash \top \Rightarrow \top$$

E-BOT
$$\Delta, \Sigma \vdash \bot \Rightarrow \bot$$

E-INT
$$\Delta, \Sigma \vdash \textbf{Int} \Rightarrow \textbf{Int}$$

E-TVAR
$$\frac{\alpha * A \in \Delta}{\Delta, \Sigma \vdash \alpha \Rightarrow \alpha}$$

E-SIG
$$\frac{X\langle \overline{\alpha}, \overline{\beta} \rangle \mapsto C \in \Delta \qquad \overline{\Sigma \vdash S \Rightarrow \langle A, B \rangle}}{\Delta, \Sigma \vdash X\langle \overline{S} \rangle \Rightarrow [\overline{A/\alpha}, \overline{B/\beta}]C}$$

E-ARR
$$\frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \Sigma \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash A \to B \Rightarrow A_1 \to B_1}$$

E-AND
$$\frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \Sigma \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash A \,\&\, B \Rightarrow A_1 \,\&\, B_1}$$

E-TRAIT
$$\frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \Sigma \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash \textbf{Trait}[A, B] \Rightarrow \textbf{Trait}[A_1, B_1]}$$

E-RCD
$$\frac{\Delta, \Sigma \vdash A \Rightarrow B}{\Delta, \Sigma \vdash \{l : A\} \Rightarrow \{l : B\}}$$

E-ALL
$$\frac{\Delta, \Sigma \vdash A \Rightarrow A_1 \qquad \Delta, \alpha * A_1 \vdash B \Rightarrow B_1}{\Delta, \Sigma \vdash \forall(\alpha * A).B \Rightarrow \forall(\alpha * A_1).B_1}$$

$\boxed{\Sigma \vdash S \Rightarrow \langle A, B \rangle}$

E-SORT1SORT
$$\frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash \alpha \Rightarrow \langle \alpha, \beta \rangle}$$

E-SORT1
$$\Sigma \vdash A \Rightarrow \langle A, A \rangle$$

E-SORT2
$$\Sigma \vdash A\%B \Rightarrow \langle A \,\&\, B, B \rangle$$

$\boxed{\Sigma \vdash^{c}_{p} A \Rightarrow B}$

TR-top
$$\Sigma \vdash^{c}_{p} \top \Rightarrow \top$$

TR-bot
$$\Sigma \vdash^{c}_{p} \bot \Rightarrow \bot$$

TR-int
$$\Sigma \vdash^{c}_{p} \textbf{Int} \Rightarrow \textbf{Int}$$

TR-positive
$$\frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash^{\text{false}}_{+} \alpha \Rightarrow \beta}$$

TR-ctrPositive
$$\frac{\alpha \mapsto \beta \in \Sigma}{\Sigma \vdash^{\text{true}}_{+} \alpha \Rightarrow \textbf{Trait}[\alpha, \beta]}$$

TR-tvar
$$\Sigma \vdash^{c}_{p} \alpha \Rightarrow \alpha$$

TR-rcd
$$\frac{\Sigma \vdash^{\text{isCapitalized}(l)}_{p} A \Rightarrow B}{\Sigma \vdash^{c}_{p} \{l : A\} \Rightarrow \{l : B\}}$$

TR-arr
$$\frac{\Sigma \vdash^{c}_{\text{flip}(p)} A \Rightarrow A_1 \qquad \Sigma \vdash^{c}_{p} B \Rightarrow B_1}{\Sigma \vdash^{c}_{p} A \to B \Rightarrow A_1 \to B_1}$$

TR-trait
$$\frac{\Sigma \vdash^{c}_{\text{flip}(p)} A \Rightarrow A_1 \qquad \Sigma \vdash^{c}_{p} B \Rightarrow B_1}{\Sigma \vdash^{c}_{p} \textbf{Trait}[A, B] \Rightarrow \textbf{Trait}[A_1, B_1]}$$

TR-and
$$\frac{\Sigma \vdash^{c}_{p} A \Rightarrow A_1 \qquad \Sigma \vdash^{c}_{p} B \Rightarrow B_1}{\Sigma \vdash^{c}_{p} A \,\&\, B \Rightarrow A_1 \,\&\, B_1}$$

TR-all
$$\frac{\Sigma \vdash^{c}_{p} A \Rightarrow A_1 \qquad \Sigma \backslash \alpha \vdash^{c}_{p} B \Rightarrow B_1}{\Sigma \vdash^{c}_{p} \forall (\alpha * A).B \Rightarrow \forall (\alpha * A_1).B_1}$$

$\boxed{A <: B}$

S-refl
$$A <: A$$

S-trans
$$\frac{A <: B \qquad B <: C}{A <: C}$$

S-topLike
$$A <: \rceil B \lceil$$

S-bot
$$\bot <: A$$

S-rcd
$$\frac{A <: B}{\{l : A\} <: \{l : B\}}$$

S-andL
$$A \,\&\, B <: A$$

S-andR
$$A \,\&\, B <: B$$

S-arr
$$\frac{B_1 <: A_1 \qquad A_2 <: B_2}{A_1 \to A_2 <: B_1 \to B_2}$$

S-and
$$\frac{A <: B \qquad A <: C}{A <: B \,\&\, C}$$

S-forall
$$\frac{B_1 <: B_2 \qquad A_2 <: A_1}{\forall (\alpha * A_1).B_1 <: \forall (\alpha * A_2).B_2}$$

S-trait
$$\frac{A_2 <: A_1 \qquad B_1 <: B_2}{\textbf{Trait}[A_1, B_1] <: \textbf{Trait}[A_2, B_2]}$$

S-distArr
$$(A \to B) \,\&\, (A \to C) <: A \to B \,\&\, C$$

S-distTrait
$$\textbf{Trait}[A, B] \,\&\, \textbf{Trait}[A, C] <: \textbf{Trait}[A, B \,\&\, C]$$

S-distRcd
$$\{l : A\} \,\&\, \{l : B\} <: \{l : A \,\&\, B\}$$

S-distAll
$$\forall (\alpha * A).B \,\&\, \forall (\alpha * A).C <: \forall (\alpha * A).B \,\&\, C$$

$\boxed{\rceil A \lceil}$

TL-top
$$\rceil \top \lceil$$

TL-and
$$\frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \,\&\, B \lceil}$$

TL-arr
$$\frac{\rceil B \lceil}{\rceil A \to B \lceil}$$

TL-rcd
$$\frac{\rceil A \lceil}{\rceil \{l : A\} \lceil}$$

TL-all
$$\frac{\rceil B \lceil}{\rceil \forall (\alpha * A).B \lceil}$$

TL-trait
$$\frac{\rceil B \lceil}{\rceil \textbf{Trait}[A, B] \lceil}$$

$\boxed{\Delta \vdash A * B}$

$$\textsc{D-topL}\quad \frac{\rceil A \lceil}{\Delta \vdash A * B} \qquad \textsc{D-topR}\quad \frac{\rceil B \lceil}{\Delta \vdash A * B} \qquad \textsc{D-arr}\quad \frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2} \qquad \textsc{D-andL}\quad \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

$$\textsc{D-andR}\quad \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2} \qquad \textsc{D-rcdEq}\quad \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}} \qquad \textsc{D-rcdNeq}\quad \frac{l_1 \neq l_2}{\Delta \vdash \{l_1 : A\} * \{l_2 : B\}}$$

$$\textsc{D-tvarL}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B} \qquad \textsc{D-tvarR}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash B * \alpha} \qquad \textsc{D-forall}\quad \frac{\Delta, \alpha * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(\alpha * A_1).B_1 * \forall(\alpha * A_2).B_2}$$

$$\textsc{D-trait}\quad \frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \textbf{Trait}[A_1, B_1] * \textbf{Trait}[A_2, B_2]} \qquad \textsc{D-traitArr}\quad \frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \textbf{Trait}[A_1, B_1] * A_2 \to B_2}$$

$$\textsc{D-arrTrait}\quad \frac{\Delta \vdash B_1 * B_2}{\Delta \vdash A_1 \to B_1 * \textbf{Trait}[A_2, B_2]} \qquad \textsc{D-ax}\quad \frac{A *_{ax} B}{\Delta \vdash A * B}$$

$\boxed{A *_{ax} B}$

$$\textsc{Dax-intArr}\quad \textbf{Int} *_{ax} A_1 \to A_2 \qquad \textsc{Dax-intRcd}\quad \textbf{Int} *_{ax} \{l : A\} \qquad \textsc{Dax-intAll}\quad \textbf{Int} *_{ax} \forall(\alpha * B_1).B_2 \qquad \textsc{Dax-intTrait}\quad \textbf{Int} *_{ax} \textbf{Trait}[A, B]$$

$$\textsc{Dax-arrAll}\quad A_1 \to A_2 *_{ax} \forall(\alpha * B_1).B_2 \qquad \textsc{Dax-arrRcd}\quad A_1 \to A_2 *_{ax} \{l : B\} \qquad \textsc{Dax-arrTrait}\quad A_1 \to A_2 *_{ax} \textbf{Trait}[A, B]$$

$$\textsc{Dax-arrAll}\quad \forall(\alpha * A_1).A_2 *_{ax} \{l : B\} \qquad \textsc{Dax-arrTrait}\quad \forall(\alpha * A_1).A_2 *_{ax} \textbf{Trait}[A, B]$$

Note: For each $A *_{ax} B$ we have a symmetric rule $B *_{ax} A$.

$\boxed{\Delta; \Gamma \vdash P \Rightarrow A \leadsto e}$

$$\textsc{T-tyDecl}\quad \frac{\begin{array}{c}\text{fresh } \overline{\beta} \qquad \Delta; \overline{\alpha \mapsto \beta} \vdash A \Rightarrow A_1 \\ \Delta; \overline{\alpha \mapsto \beta} \vdash B \Rightarrow B_1 \qquad \overline{\alpha \mapsto \beta} \vdash^{\text{false}}_{+} B_1 \Rightarrow B_2 \qquad \Delta, X\langle \overline{\alpha}, \overline{\beta} \rangle \mapsto A_1 \,\&\, B_2; \Gamma \vdash P \Rightarrow C \leadsto e\end{array}}{\Delta; \Gamma \vdash \textbf{type}\ X\langle \overline{\alpha} \rangle\ \textbf{extends}\ A = B; P \Rightarrow C \leadsto e}$$

$$\textsc{T-tmDecl}\quad \frac{\Delta; \Gamma \vdash E \Rightarrow A \leadsto e_1 \qquad \Delta; \Gamma, x : A \vdash P \Rightarrow B \leadsto e_2}{\Delta; \Gamma \vdash x = E; P \Rightarrow B \leadsto \textbf{let}\ x : |A| = e_1\ \textbf{in}\ e_2}$$

$\boxed{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e}$

T-TOP

$$\Delta;\Gamma \vdash \top \Rightarrow \top \rightsquigarrow \top$$

T-NAT

$$\Delta;\Gamma \vdash i \Rightarrow \textbf{Int} \rightsquigarrow i$$

T-VAR

$$\frac{(x : A) \in \Gamma}{\Delta;\Gamma \vdash x \Leftarrow A \rightsquigarrow x}$$

T-APP

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \rightarrow A_2 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A_1 \rightsquigarrow e_2}{\Delta;\Gamma \vdash E_1\ E_2 \Rightarrow A_2 \rightsquigarrow e_1\ e_2}$$

T-ANNO

$$\frac{\Delta;\bullet \vdash A \Rightarrow B \qquad \Delta;\Gamma \vdash E \Leftarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash E : A \Rightarrow B \rightsquigarrow e : |B|}$$

T-RCD

$$\frac{\Delta;\Gamma \vdash E \Rightarrow A \rightsquigarrow e}{\Delta;\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \rightsquigarrow e}$$

T-PROJ

$$\frac{\Delta;\Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta;\Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}$$

T-TABS

$$\frac{\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta, \alpha * A_1;\Gamma \vdash E \Rightarrow B \rightsquigarrow e}{\Delta;\Gamma \vdash \Lambda(\alpha * A).E \Rightarrow \forall(\alpha * |A_1|).B \rightsquigarrow e}$$

T-TAPP

$$\frac{\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta;\Gamma \vdash E \Rightarrow \forall(\alpha * B).C \rightsquigarrow e \qquad \Delta \vdash A_1 * B}{\Delta;\Gamma \vdash E\ @A \Rightarrow [A_1/\alpha]C \rightsquigarrow e\ |A_1|}$$

T-LET

$$\frac{\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta;\Gamma, x : A_1 \vdash E_1 \Leftarrow A_1 \rightsquigarrow e_1 \qquad \Delta;\Gamma, x : A_1 \vdash E_2 \Rightarrow B \rightsquigarrow e_2}{\Delta;\Gamma \vdash \textbf{let}\ x : A = E_1\ \textbf{in}\ E_2 \Rightarrow B \rightsquigarrow \textbf{let}\ x : |A_1| = e_1\ \textbf{in}\ e_2}$$

T-MERGETRAIT

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \textbf{Trait}[A_1, B_1] \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow \textbf{Trait}[A_2, B_2] \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2}{\Delta;\Gamma \vdash E_1\ ,,\ E_2 \Rightarrow \textbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2] \rightsquigarrow \lambda(\textbf{self} : |A_1 \,\&\, A_2|).\ e_1\ \textbf{self}\ ,,\ e_2\ \textbf{self}}$$

T-MERGE

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Delta;\Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad \Delta \vdash A_1 * A_2}{\Delta;\Gamma \vdash E_1\ ,,\ E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow e_1\ ,,\ e_2}$$

T-NEW

$$\frac{\Delta;\Gamma \vdash E \Rightarrow \textbf{Trait}[A, B] \rightsquigarrow e \qquad B <: A}{\Delta;\Gamma \vdash \textbf{new}\ E \Rightarrow B \rightsquigarrow \textbf{let self} : |B| = e\ \textbf{self in self}}$$

T-OPEN

$$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \overline{\{l_i : A_i\}} \rightsquigarrow e_1 \qquad \Delta;\Gamma, \overline{l_i : A_i} \vdash E_2 \Rightarrow e_2 \rightsquigarrow B}{\Delta;\Gamma \vdash \textbf{open}\ E_1\ \textbf{in}\ E_2 \Rightarrow B \rightsquigarrow \textbf{let}\ x = e_1\ \textbf{in}\ \overline{\textbf{let}\ l_i : |A| = x.l_i}\ \textbf{in}\ e_2}$$

T-TRAIT

$$\frac{\begin{array}{c}\Delta; \bullet \vdash A \Rightarrow A_1 \qquad \Delta; \bullet \vdash B \Rightarrow B_1 \qquad \Delta; \Gamma, \mathbf{self} : A_1 \vdash E_1 \Rightarrow \mathbf{Trait}[A_2, B_2] \rightsquigarrow e_1 \\ A_1 <: A_2 \qquad \Delta; \Gamma, \mathbf{self} : A_1, \mathbf{super} : B_2 \vdash E_2 \Rightarrow C \rightsquigarrow e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1\end{array}}{\begin{array}{c}\Delta; \Gamma \vdash \mathbf{trait}[\mathbf{self} : A] \ \mathbf{implements}\ B\ \mathbf{inherits}\ E_1 \ \texttt{=>}\ E_2 \Rightarrow \mathbf{Trait}[A_1, C \,\&\, B_2] \\ \rightsquigarrow \lambda(\mathbf{self} : |A_1|).\ \mathbf{let\ super} = e_1\ \mathbf{self\ in}\ e_2\ ,,\ \mathbf{super}\end{array}}$$

T-FORWARD

$$\frac{\Delta; \Gamma \vdash E_1 \Rightarrow \mathbf{Trait}[A, B] \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Leftarrow A \rightsquigarrow e_2}{\Delta; \Gamma \vdash E_1 \hat{} E_2 \Rightarrow B \rightsquigarrow e_1\ e_2}$$

$\boxed{\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e}$

T-ABS

$$\frac{\Delta; \Gamma, x : A \vdash E \Leftarrow B \rightsquigarrow e}{\Delta; \Gamma \vdash \lambda x.E \Leftarrow A \to B \rightsquigarrow \lambda x.E}$$

T-SUB

$$\frac{\Delta; \Gamma \vdash E \Rightarrow B \rightsquigarrow e \qquad B <: A}{\Delta; \Gamma \vdash E \Leftarrow A \rightsquigarrow e}$$

$$\begin{aligned} |\mathbf{Trait}[A, B]| &= |A| \to |B| \\ |A \to B| &= |A| \to |B| \\ |A \,\&\, B| &= |A| \,\&\, |B| \\ |\{l : A\}| &= \{l : |A|\} \\ |\forall(\alpha * A).B| &= \forall(\alpha * |A|).|B| \\ |A| &= A \end{aligned}$$

Appendix **B**

Metatheory of CP

Lemma 2.5 *[Well-formedness preservation] If $\Delta, \Sigma \vdash A \Rightarrow B$ then $|\Delta| \vdash |B|$.*

Proof. By simple induction on the derivation of the judgment. □

Lemma 2.6 *[Disjointness axiom preservation] If $A *_{ax} B$ then $|A| *_{ax} |B|$.*

Proof. Note that $|\textbf{Trait}[A, B]| = |A| \to |B|$; the rest are immediate. □

Lemma 2.7 *[Subtyping preservation] If $A <: B$ then $|A| <: |B|$.*

Proof. Most of them are just F_i^+ subtyping. We only show the rule S-TRAIT

$$\textsc{S-trait}\quad \frac{A_2 <: A_1 \qquad B_1 <: B_2}{\textbf{Trait}[A_1, B_1] <: \textbf{Trait}[A_2, B_2]}$$

$\lvert A_2\rvert <: \lvert A_1\rvert$	By i.h.
$\lvert B_1\rvert <: \lvert B_2\rvert$	By i.h.
$\lvert A_1\rvert \to \lvert B_1\rvert <: \lvert A_2\rvert \to \lvert B_2\rvert$	By TS-ARR

□

Lemma 2.8 *[Disjointness preservation] If $\Delta \vdash A * B$ then $|\Delta| \vdash |A| * |B|$.*

Proof. By induction on the derivation of the judgment.

- D-TOPL, D-TOPR, and D-RCDNEQ are immediate.
- $$\textsc{D-tvarL}\quad \frac{(\alpha * A) \in \Delta \qquad A <: B}{\Delta \vdash \alpha * B}$$

$\lvert A\rvert <: \lvert B\rvert$	By Lemma 2.7
$\alpha * A \in \Delta$	Given
$\alpha * \lvert A\rvert \in \lvert\Delta\rvert$	Above
$\lvert\Delta\rvert \vdash \alpha * \lvert B\rvert$	By TD-TVARL

- $$\textsc{D-tvarR}\quad \frac{(a * A) \in \Delta \qquad A <: B}{\Delta \vdash B * a}$$

$\lvert A\rvert <: \lvert B\rvert$	By Lemma 2.7
$a * A \in \Delta$	Given
$a * \lvert A\rvert \in \lvert\Delta\rvert$	Above
$\lvert\Delta\rvert \vdash \lvert B\rvert * a$	By TD-tvarR

- $$\textsc{D-forall}\quad \frac{\Delta, a * A_1 \,\&\, A_2 \vdash B_1 * B_2}{\Delta \vdash \forall(a * A_1).B_1 * \forall(a * A_2).B_2}$$

$\lvert\Delta\rvert, a * \lvert A_1\rvert \,\&\, \lvert A_2\rvert \vdash \lvert B_1\rvert * \lvert B_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \forall(a * \lvert A_1\rvert).B_1 * \forall(a * \lvert A_2\rvert).\lvert B_2\rvert$	By TD-forall

- $$\textsc{D-rcdEq}\quad \frac{\Delta \vdash A * B}{\Delta \vdash \{l : A\} * \{l : B\}}$$

$\lvert\Delta\rvert \vdash \lvert A\rvert * \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \{l : \lvert A\rvert\} * \{l : \lvert B\rvert\}$	By TD-rcdEq

- $$\textsc{D-arr}\quad \frac{\Delta \vdash A_2 * B_2}{\Delta \vdash A_1 \to A_2 * B_1 \to B_2}$$

$\lvert\Delta\rvert \vdash \lvert A_2\rvert * \lvert B_2\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert A_1\rvert \to \lvert A_2\rvert * \lvert B_1\rvert \to \lvert B_2\rvert$	By TD-arr

- $$\textsc{D-andL}\quad \frac{\Delta \vdash A_1 * B \qquad \Delta \vdash A_2 * B}{\Delta \vdash A_1 \,\&\, A_2 * B}$$

$\lvert\Delta\rvert \vdash \lvert A_1\rvert * \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert A_2\rvert * \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert \vdash \lvert A_1\rvert \,\&\, \lvert A_2\rvert * \lvert B\rvert$	By TD-andL

- $$\textsc{D-andR}\quad \frac{\Delta \vdash A * B_1 \qquad \Delta \vdash A * B_2}{\Delta \vdash A * B_1 \,\&\, B_2}$$

$|\Delta| \vdash |A| * |B_1|$ By i.h.

$|\Delta| \vdash |A| * |B_2|$ By i.h.

$|\Delta| \vdash |A| * |B_1| \,\&\, |B_2|$ By TD-andR

- $$\textsc{D-trait}\quad \frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \textbf{Trait}[A_1, B_1] * \textbf{Trait}[A_2, B_2]}$$

$|\Delta| \vdash |B_1| * |B_2|$ By i.h.

$|\Delta| \vdash |A_1| \rightarrow |B_1| * |A_2| \rightarrow |B_2|$ By TD-arr

- $$\textsc{D-traitArr}\quad \frac{\Delta \vdash B_1 * B_2}{\Delta \vdash \textbf{Trait}[A_1, B_1] * A_2 \rightarrow B_2}$$

$|\Delta| \vdash |B_1| * |B_2|$ By i.h.

$|\Delta| \vdash |A_1| \rightarrow |B_1| * |A_2| \rightarrow |B_2|$ By TD-arr

- $$\textsc{D-arrTrait}\quad \frac{\Delta \vdash B_1 * B_2}{\Delta \vdash A_1 \rightarrow B_1 * \textbf{Trait}[A_2, B_2]}$$

$|\Delta| \vdash |B_1| * |B_2|$ By i.h.

$|\Delta| \vdash |A_1| \rightarrow |B_1| * |A_2| \rightarrow |B_2|$ By TD-arr

- $$\textsc{D-ax}\quad \frac{A *_{ax} B}{\Delta \vdash A * B}$$

$|A| *_{ax} |B|$ By Lemma 2.6

$|\Delta| \vdash |A| * |B|$ By TD-ax

□

Theorem 2.3 *[Type-safety] We have that:*

- *If* $\Delta; \Gamma \vdash P \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.
- *If* $\Delta; \Gamma \vdash E \Rightarrow A \rightsquigarrow e$ *then* $|\Delta|; |\Gamma| \vdash e \Rightarrow |A|$.

- *If* $\Delta;\Gamma \vdash E \Leftarrow A \leadsto e$ *then* $|\Delta|;|\Gamma| \vdash e \Leftarrow |A|$.

Proof. By induction on the typing judgment.

- $$\textsc{T-tyDecl}\quad \frac{\begin{array}{c}\text{fresh } \overline{\beta} \qquad \Delta;\overline{\alpha \mapsto \beta} \vdash A \Rightarrow A_1 \qquad \Delta;\overline{\alpha \mapsto \beta} \vdash B \Rightarrow B_1 \\ \overline{\alpha \mapsto \beta} \vdash^{\text{false}}_{+} B_1 \Rightarrow B_2 \qquad \Delta, X\langle\overline{\alpha},\overline{\beta}\rangle \mapsto A_1 \,\&\, B_2;\Gamma \vdash P \Rightarrow C \leadsto e\end{array}}{\Delta;\Gamma \vdash \textbf{type } X\langle\overline{\alpha}\rangle \textbf{ extends } A = B; P \Rightarrow C \leadsto e}$$

$$\begin{array}{ll} |\Delta|;|\Gamma| \vdash e \Rightarrow |C| & \text{By i.h.} \end{array}$$

- $$\textsc{T-tmDecl}\quad \frac{\Delta;\Gamma \vdash E \Rightarrow A \leadsto e_1 \qquad \Delta;\Gamma, x : A \vdash P \Rightarrow B \leadsto e_2}{\Delta;\Gamma \vdash x = E; P \Rightarrow B \leadsto \textbf{let } x : |A| = e_1 \textbf{ in } e_2}$$

$$\begin{array}{ll} |\Delta|;|\Gamma| \vdash e_1 \Rightarrow |A| & \text{By i.h.} \\ |\Delta|;|\Gamma|, x : |A| \vdash e_2 \Rightarrow |B| & \text{By i.h.} \\ |\Delta|;|\Gamma| \vdash \textbf{let } x : |A| = e_1 \textbf{ in } e_2 \Rightarrow |B| & \text{By TT-}\textsc{let} \end{array}$$

- T-TOP, T-NAT, and T-VAR are immediate.

- $$\textsc{T-app}\quad \frac{\Delta;\Gamma \vdash E_1 \Rightarrow A_1 \to A_2 \leadsto e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A_1 \leadsto e_2}{\Delta;\Gamma \vdash E_1\ E_2 \Rightarrow A_2 \leadsto e_1\ e_2}$$

$$\begin{array}{ll} |\Delta|;|\Gamma| \vdash e_1 \Rightarrow |A_1| \to |A_2| & \text{By i.h.} \\ |\Delta|;|\Gamma| \vdash e_2 \Leftarrow |A_2| & \text{By i.h.} \\ |\Delta|;|\Gamma| \vdash e_1\ e_2 \Rightarrow |A_2| & \text{By TT-}\textsc{app} \end{array}$$

- $$\textsc{T-anno}\quad \frac{\Delta;\bullet \vdash A \Rightarrow B \qquad \Delta;\Gamma \vdash E \Leftarrow B \leadsto e}{\Delta;\Gamma \vdash E : A \Rightarrow B \leadsto e : |B|}$$

$$\begin{array}{ll} |\Delta|;|\Gamma| \vdash e \Leftarrow |B| & \text{By i.h.} \\ |\Delta|;|\Gamma| \vdash e : |B| \Rightarrow |B| & \text{By TT-}\textsc{anno} \end{array}$$

- $$\textsc{T-rcd}\quad \frac{\Delta;\Gamma \vdash E \Rightarrow A \leadsto e}{\Delta;\Gamma \vdash \{l = E\} \Rightarrow \{l : A\} \leadsto e}$$

$$\begin{array}{ll} |\Delta|;|\Gamma| \vdash e \Rightarrow |A| & \text{By i.h.} \\ |\Delta|;|\Gamma| \vdash \{l = e\} \Rightarrow \{l : |A|\} & \text{By TT-}\textsc{rcd} \end{array}$$

- $$\frac{\Delta ; \Gamma \vdash E \Rightarrow \{l : A\} \rightsquigarrow e}{\Delta ; \Gamma \vdash E.l \Rightarrow A \rightsquigarrow e}\ \textsc{T-proj}$$

$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash e \Rightarrow \{l : \lvert A\rvert\}$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash e.l \Rightarrow \lvert A\rvert$	By TT-proj

- $$\frac{\Delta ; \bullet \vdash A \Rightarrow A_1 \qquad \Delta, \alpha * A_1 ; \Gamma \vdash E \Rightarrow B \rightsquigarrow e}{\Delta ; \Gamma \vdash \Lambda(\alpha * A).E \Rightarrow \forall(\alpha * \lvert A_1\rvert).B \rightsquigarrow e}\ \textsc{T-tabs}$$

$\lvert\Delta\rvert \vdash \lvert A_1\rvert$	By Lemma 2.5
$\lvert\Delta\rvert ; \lvert\Gamma\rvert, \alpha * \lvert A\rvert \vdash e \Rightarrow \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash \Lambda(\alpha * \lvert A\rvert).\, e \Rightarrow \forall(\alpha * \lvert A\rvert).\, \lvert B\rvert$	By TT-tabs

- $$\frac{\Delta ; \bullet \vdash A \Rightarrow A_1 \qquad \Delta ; \Gamma \vdash E \Rightarrow \forall(\alpha * B).C \rightsquigarrow e \qquad \Delta \vdash A_1 * B}{\Delta ; \Gamma \vdash E \ @A \Rightarrow [A_1/\alpha]C \rightsquigarrow e \ \lvert A_1\rvert}\ \textsc{T-tapp}$$

$\lvert\Delta\rvert \vdash \lvert A_1\rvert$	By Lemma 2.5
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash e \Rightarrow \forall(\alpha * \lvert B\rvert).\lvert C\rvert$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash \lvert A_1\rvert * \lvert B\rvert$	By Lemma 2.8
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash e \ \lvert A\rvert \Rightarrow [\lvert A_1\rvert/\alpha]\lvert C\rvert$	By TT-tapp

- $$\frac{\Delta ; \bullet \vdash A \Rightarrow A_1 \qquad \Delta ; \Gamma, x : A_1 \vdash E_1 \Leftarrow A_1 \rightsquigarrow e_1 \qquad \Delta ; \Gamma, x : A_1 \vdash E_2 \Rightarrow B \rightsquigarrow e_2}{\Delta ; \Gamma \vdash \mathbf{let}\ x : A = E_1\ \mathbf{in}\ E_2 \Rightarrow B \rightsquigarrow \mathbf{let}\ x : \lvert A_1\rvert = e_1\ \mathbf{in}\ e_2}\ \textsc{T-let}$$

$\lvert\Delta\rvert \vdash \lvert A_1\rvert$	By Lemma 2.5
$\lvert\Delta\rvert ; \lvert\Gamma\rvert, x : \lvert A_1\rvert \vdash e_1 \Leftarrow \lvert A\rvert$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert, x : \lvert A_1\rvert \vdash e_2 \Rightarrow \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash \mathbf{let}\ x : \lvert A_1\rvert = E_1\ \mathbf{in}\ E_2 \Rightarrow \lvert B\rvert$	By TT-let

- $$\frac{\Delta ; \Gamma \vdash E_1 \Rightarrow \mathbf{Trait}[A_1, B_1] \rightsquigarrow e_1 \qquad \Delta ; \Gamma \vdash E_2 \Rightarrow \mathbf{Trait}[A_2, B_2] \rightsquigarrow e_2 \qquad \Delta \vdash B_1 * B_2}{\Delta ; \Gamma \vdash E_1 \ ,,\ E_2 \Rightarrow \mathbf{Trait}[A_1 \,\&\, A_2, B_1 \,\&\, B_2] \rightsquigarrow \lambda(\mathbf{self} : \lvert A_1 \,\&\, A_2\rvert).\, e_1\ \mathbf{self} \ ,,\ e_2\ \mathbf{self}}\ \textsc{T-mergeTrait}$$

$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert A_1 \to B_1\rvert$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert \vdash e_2 \Rightarrow \lvert A_2 \to B_2\rvert$	By i.h.
$\lvert\Delta\rvert ; \lvert\Gamma\rvert, \mathbf{self} : \lvert A_1 \,\&\, A_2\rvert \vdash \mathbf{self} \Rightarrow \lvert A_1 \,\&\, A_2\rvert$	By TT-var

$\lvert A_1 \,\&\, A_2\rvert <: \lvert A_1\rvert$	By TS-ANDL
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \textbf{self} \Leftarrow \lvert A_1\rvert$	By TT-SUB
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1 \; \textbf{self} \Rightarrow \lvert B_1\rvert$	By TT-APP
$\lvert A_1 \,\&\, A_2\rvert <: \lvert A_2\rvert$	By TS-ANDR
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \textbf{self} \Leftarrow \lvert A_2\rvert$	By TT-SUB
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_2 \; \textbf{self} \Rightarrow \lvert B_2\rvert$	By TT-APP
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1 \; \textbf{self} \;,,\; e_2 \; \textbf{self} \Rightarrow \lvert B_1 \,\&\, B_2\rvert$	By TT-MERGE
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \lambda(\textbf{self} : \lvert A_1 \,\&\, A_2\rvert).\, e_1 \; \textbf{self} \;,,\; e_2 \; \textbf{self} \Rightarrow \lvert A_1 \,\&\, A_2\rvert \to \lvert B_1 \,\&\, B_2\rvert$	By TT-ABS

- $$\textsc{T-merge}\quad \frac{\Delta; \Gamma \vdash E_1 \Rightarrow A_1 \rightsquigarrow e_1 \qquad \Delta; \Gamma \vdash E_2 \Rightarrow A_2 \rightsquigarrow e_2 \qquad \Delta \vdash A_1 * A_2}{\Delta; \Gamma \vdash E_1 \;,,\; E_2 \Rightarrow A_1 \,\&\, A_2 \rightsquigarrow e_1 \;,,\; e_2}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert A\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_2 \Rightarrow \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \lvert A\rvert * \lvert B\rvert$	By Lemma 2.8
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1 ,, e_2 \Rightarrow \lvert A\rvert \,\&\, \lvert B\rvert$	By TT-MERGE

- $$\textsc{T-new}\quad \frac{\Delta; \Gamma \vdash E \Rightarrow \textbf{Trait}[A, B] \rightsquigarrow e \qquad B <: A}{\Delta; \Gamma \vdash \textbf{new}\; E \Rightarrow B \rightsquigarrow \textbf{let self} : \lvert B\rvert = e \; \textbf{self in self}}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e \Rightarrow \lvert A\rvert \to \lvert B\rvert$	By i.h.
$\lvert B\rvert <: \lvert A\rvert$	By Lemma 2.7
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \textbf{self} : \lvert B\rvert \vdash \textbf{self} \Rightarrow \lvert B\rvert$	By TT-VAR
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \textbf{self} : \lvert B\rvert \vdash \textbf{self} \Leftarrow \lvert A\rvert$	By TT-SUB
$\lvert\Delta\rvert; \lvert\Gamma\rvert, \textbf{self} : \lvert B\rvert \vdash e \; \textbf{self} \Leftarrow \lvert B\rvert$	By TT-APP
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \textbf{let self} : \lvert B\rvert = e \; \textbf{self in self} \Rightarrow \lvert B\rvert$	By TT-LET

- $$\textsc{T-open}\quad \frac{\Delta; \Gamma \vdash E_1 \Rightarrow \{\overline{l_i : A_i}\} \rightsquigarrow e_1 \qquad \Delta; \Gamma, \overline{l_i : A_i} \vdash E_2 \Rightarrow e_2 \rightsquigarrow B}{\Delta; \Gamma \vdash \textbf{open}\; E_1 \;\textbf{in}\; E_2 \Rightarrow B \rightsquigarrow \textbf{let}\; x = e_1 \;\textbf{in}\; \overline{\textbf{let}\; l_i : \lvert A\rvert = x.l_i} \;\textbf{in}\; e_2}$$

$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert\{\overline{l_i : A_i}\}\rvert$	By i.h.
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash e_1.l_i \Rightarrow \lvert A_i\rvert$	By TT-PROJ
$\lvert\Delta\rvert; \lvert\Gamma\rvert \vdash \overline{\textbf{let}\; l_i : \lvert A_i\rvert = e_1.l_i} \;\textbf{in}\; e_2 \Rightarrow \lvert B\rvert$	By TT-LET

- $$\frac{\Delta;\bullet \vdash A \Rightarrow A_1 \qquad \Delta;\bullet \vdash B \Rightarrow B_1 \qquad \Delta;\Gamma,\textbf{self} : A_1 \vdash E_1 \Rightarrow \textbf{Trait}[A_2, B_2] \leadsto e_1 \\ A_1 <: A_2 \qquad \Delta;\Gamma,\textbf{self} : A_1, \textbf{super} : B_2 \vdash E_2 \Rightarrow C \leadsto e_2 \qquad C * B_2 \qquad C \,\&\, B_2 <: B_1}{\Delta;\Gamma \vdash \textbf{trait}[\textbf{self} : A]\ \textbf{implements}\ B\ \textbf{inherits}\ E_1\ \texttt{=>}\ E_2 \Rightarrow \textbf{Trait}[A_1, C \,\&\, B_2] \\ \leadsto \text{ƛ}(\textbf{self}\!:\!|A_1|).\ \textbf{let super} = e_1\ \textbf{self in}\ e_2\ ,,\ \textbf{super}}\ \textsc{T-trait}$$

$\lvert\Delta\rvert \vdash \lvert A_1\rvert$	By Lemma 2.5
$\lvert\Delta\rvert \vdash \lvert B_1\rvert$	By Lemma 2.5
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \textbf{self} : \lvert A_1\rvert \vdash e_1 \Rightarrow \lvert A_2\rvert \to \lvert B_2\rvert$	By i.h.
$\lvert A_1\rvert <: \lvert A_2\rvert$	By Lemma 2.7
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \textbf{self} : \lvert A_1\rvert, \textbf{super} : \lvert B_2\rvert \vdash e_2 \Rightarrow \lvert C\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \textbf{self} : \lvert A_1\rvert \vdash \textbf{self} \Rightarrow \lvert A_1\rvert$	By TT-var
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \textbf{self} : \lvert A_1\rvert \vdash \textbf{self} \Leftarrow \lvert B_1\rvert$	By TT-sub
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \textbf{self} : \lvert A_1\rvert \vdash e_1\ \textbf{self} \Rightarrow \lvert B_2\rvert$	By TT-tapp
$\lvert\Delta\rvert;\lvert\Gamma\rvert, \textbf{self} : \lvert A_1\rvert, \textbf{super} : \lvert B_2\rvert \vdash e_2\ ,,\ \textbf{super} \Rightarrow \lvert C\rvert \,\&\, \lvert B_2\rvert$	By TT-merge
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \textbf{let super} = e_1\ \textbf{self in}\ e_2\ ,,\ \textbf{super} \Rightarrow \lvert C\rvert \,\&\, \lvert B_2\rvert$	By TT-let
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \text{ƛ}(\textbf{self}\!:\!\lvert A_1\rvert).\ \textbf{let super} = e_1\ \textbf{self in}\ e_2\ ,,\ \textbf{super} \Rightarrow \lvert A_1\rvert \to (\lvert C\rvert \,\&\, \lvert B_2\rvert)$	By TT-abs

- $$\frac{\Delta;\Gamma \vdash E_1 \Rightarrow \textbf{Trait}[A, B] \leadsto e_1 \qquad \Delta;\Gamma \vdash E_2 \Leftarrow A \leadsto e_2}{\Delta;\Gamma \vdash E_1\hat{\ }E_2 \Rightarrow B \leadsto e_1\ e_2}\ \textsc{T-forward}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1 \Rightarrow \lvert A\rvert \to \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_2 \Leftarrow \lvert A\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e_1\ e_2$	By TT-app

- $$\frac{\Delta;\Gamma, x : A \vdash E \Leftarrow B \leadsto e}{\Delta;\Gamma \vdash \text{ƛ}x.E \Leftarrow A \to B \leadsto \text{ƛ}x.E}\ \textsc{T-abs}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert, x : \lvert A\rvert \vdash e \Leftarrow \lvert B\rvert$	By i.h.
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash \text{ƛ}(x\!:\!e).\ \Leftarrow \lvert A\rvert \to \lvert B\rvert$	By TT-abs

- $$\frac{\Delta;\Gamma \vdash E \Rightarrow B \leadsto e \qquad B <: A}{\Delta;\Gamma \vdash E \Leftarrow A \leadsto e}\ \textsc{T-sub}$$

$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e \Rightarrow \lvert B\rvert$	By i.h.
$\lvert B\rvert <: \lvert A\rvert$	By Lemma 2.7
$\lvert\Delta\rvert;\lvert\Gamma\rvert \vdash e \Leftarrow \lvert B\rvert$	By TT-sub

□