Reusability and Modularity in Consumer, Producer, and Transformation Operations

Haoyuan Zhang
Department of Computer Science
The University of Hong Kong

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

June 2019

Abstract of thesis entitled

"Reusability and Modularity in Consumer, Producer, and Transformation Operations"

Submitted by

Haoyuan Zhang

for the degree of Doctor of Philosophy
at The University of Hong Kong
in June 2019

Since 1954 when the first high-level programming language FORTRAN was invented, the past decades have witnessed several significant revolutions of modularization in programming areas: structured programming, modules and encapsulation, abstract objects and datatypes, inheritance, and many other paradigms of type systems and recursion patterns. The essence of modularization is that, by separating the functionality of a program into a network of independent or loosely dependent components, each component is easier to be reused and maintained without affecting others. And the ultimate goal of modularization, is to enhance the efficiency/productivity of programmers and reduce the cost of software development and maintenance, and meanwhile ensure (type) safety and execution performance as much as possible. Newly proposed modularization techniques in existing languages, can in turn promote the development of new language features and even new language paradigms.

In recent few years, researchers have put great effort on resolving the Expression Problem (coined by Philip Wadler) in different languages. The requirements for modularization are well described in the Expression Problem, especially it highlights two dimensions of extensibility at the same time: on data variants and operations. Unfortunately, although existing approaches satisfy this primary requirement, the term "operation" is not well identified; more specifically, most existing approaches focus on the modularity of consumer operations that consume data structures and collect information, but not producer operations that build data structures. Additionally, it is difficult to balance modularity with other attributes, including type-safety, conciseness, expressiveness, and efficiency.

This dissertation investigates modularization techniques for consumers and producers with simple, type-safe and reusable patterns, without breaking the modularity of data variants. Specifically, three research problems are studied in this dissertation: boilerplate traversals, type-safe modular parsing and modular unfolds. As a result, this dissertation proposes: *Shy*, a Java library that captures generic traversal patterns and automatically generates boilerplate traversal code with extensibility; a type-safe pattern for writing modular parsers in object-oriented programming; and *SCCL*, a specialized coalgebra combinator library for modularizing producers by modularizing categorical f-coalgebras in Haskell. Consequently, there is a significant reduction in client code due to modularity and reusability, evidenced by a number of case studies and applications.

(*An abstract of 351 words*)

To my beloved parents

Declaration

I declare that this thesis represents my own work, except where due acknowledgement 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.

.....

Haoyuan Zhang

June 2019

Acknowledgements

First of all, I would like to give my sincere gratitude to my supervisor, Dr. Bruno C. d. S. Oliveira, for his continued teaching, support and encouragement throughout my PhD study. In the early months of my candidature, Dr. Oliveira led me to the area of programming languages step by step, from simple tasks to complex projects and research problems, and helped me a lot with paper writing. He set a very good image of being an independent researcher. It is my great honor being his student, I feel fully respected, encouraged and appreciate a lot for his patient guidance in my research exploration.

I also feel fortunate having Prof. T. H. Tse as my co-supervisor. He provided many valuable comments and feedback on my research work, paper and thesis writing, and especially I appreciate his attendance in my probation and pre-graduation talk, where he showed his seriousness with challenging questions, but soon gave more suggestions and encouraged me to be confident.

I feel grateful to all my colleagues in the Programming Languages group at HKU, including (in no particular order): Zhiyuan Shi, Tomas Tauber, Zewei Chu, Huang Li, Xuan Bi, Weixin Zhang, Yanlin Wang, Yanpeng Yang, Ningning Xie, Jinxu Zhao, Xuejing Huang and Yaoda Zhou. My special thanks to professors Tijs van der Storm, Marco Servetto, and Shin-Cheng Mu for the collaboration in many research projects.

I would also like to thank Prof. Kenneth Wong, Prof. Cho-Li Wang and Prof. Zhenjiang Hu for joining my defense examination and providing valuable comments on revision, and thank Prof. Ngai Wong for being TEC chair.

Last but not least, I feel grateful to my parents for loving, supporting and respecting me unconditionally. Thank you for bringing me to the world. I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recently on *Hacker News*[1], a social news website for the community of programmers, a user shared his life in Oracle under the topic "*What's the largest amount of bad code you have ever seen work?*":

> *"Oracle Database 12.2. It is close to 25 million lines of C code... Very complex pieces of logic, memory management, context switching, etc. are all held together with thousands of flags. The whole code is ridden with mysterious macros that one cannot decipher without picking a notebook and expanding relevant pats of the macros by hand. It can take a day to two days to really understand what a macro does... It takes 6 months to a year (sometimes two years!) to develop a single small feature..."*

It is unfortunate yet unsurprising to see this phenomenon, especially in the development of large software systems. Programming on millions of spaghetti code is rather cruel to engineers, and as a consequence, adding a single extension may bring disastrous effects to the entire program. When the fast growth of users and demands cannot be satisfied, extra budget has to be put to refactor old code, hopefully improving the efficiency of software development.

People have long been conscious of the importance of *modularization* in programming. It is widely acknowledged that modularity leads to code reusability, extensibility, comprehensibility, and maintainability, and essentially to the enhancement of programming efficiency as well as the reduction of development cost. The initial architecture of a program takes the responsibility for separating the functionality into a network of independent, or loosely dependent components. Modularization enforces logical boundaries among the components, consequently each component can be developed individually, and is even interchangeable without affecting other modules.

The past few decades have witnessed different manifestations of modularization along with the evolution of programming language paradigms. The idea of modularity can be traced back to the modular programming movement in around 1960s [Dijkstra, 1968].

1.1 Modular Programming: A Brief History

Since 1954 when the first high-level programming language FORTRAN [Backus, 1954] was invented, the development of various programming languages and paradigms started to grow explosively. No-

[1] `https://news.ycombinator.com/item?id=18442941`

tably the `GOTO` syntax, which still appears in some modern languages, offered flexible control of execution at that time. However, the 1960s have experienced a considerable debate on the use of `GOTO`[1]. Böhm and Jacopini [1966] proved the structured program theorem, stating that every computable algorithm can be represented by a combination of only three control structures: sequence, selection, and iteration. While Böhm and Jacopini showed the unnecessity of `GOTO`, Dijkstra [1968] was a firm and iconic opponent of `GOTO`:

> *"... our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do... our utmost to shorten the conceptual gap between the static program and the dynamic process..."*

Additionally, he coined the term "*structured programming*", which basically represents the modular programming movement at that time with a sharp decrease of `GOTO` uses.

Later Parnas [1972] promoted modularization to a new direction, namely by introducing modules that encapsulate major calculations, and hide some information from clients but only reveal the necessary interfaces to access them. The criterion, *"its interface or definition was chosen to reveal as little as possible about its inner workings"*, implies the concept of abstract *objects* with state.

Liskov and Zilles [1974] proposed *abstract data types* (ADTs) as a data abstraction technique, which has significant impact on *object-oriented programming* (OOP) until nowadays. An abstract data type captures the fundamental properties of a class of abstract objects, and those properties are precisely the operations/behaviors that people are concerned about, while concrete implementations are hidden. In OOP, ADTs are referred to as classes, and additional mechanisms including inheritance, dynamic dispatch and polymorphism further facilitate code reuse. Design patterns [Gamma, 1995], on the other hand, provide specifications for the abstraction of certain systems by utilizing existing language features.

More recently, the area has been broadened to *functional programming* (FP), a different paradigm that takes functions as the essential elements. Functional programs draw a strong connection to the mathematical logic from function abstraction and application, originating in *lambda calculus* [Church, 1932]. More importantly, *higher-order functions* and *lazy evaluation* are powerful techniques to the modularity of code, as argued by Hughes [1989]. As a result, functional programs are usually small in magnitude. The connection to mathematical theory, and elimination of side effects (by modifying state) in addition, make programs easier to reason about and predict.

As two representative programming paradigms at present, OOP and FP have different biases in modularization. OOP has tendency in *data* abstraction and reuse, while FP naturally emphasizes on the modularity of *functions/operations*. Discussion on a single paradigm can be lopsided, since its advantages in modularity derive from the infrastructure and theoretical foundations. Furthermore, *isomorphisms* are ubiquitous among the modularization patterns or abstractions in different languages. This dissertation is hence inspired to involve both OOP and FP, and to look into their internal relations.

In the late twentieth century, the perceived need for modularity on both data abstractions and operations on data structures was already in folklore, until it was coined by Wadler [1998] as the

[1] The timeline is partly referred to the course "Introduction to Programming Systems" by Dondero [2014] at Princeton University.

"Expression Problem". To quote from his post:

> *"The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts). For the concrete example, we take expressions as the data type, begin with one case (constants) and one function (evaluators), then add one more construct (plus) and one more function (conversion to a string)..."*

To summarize, a solution to the Expression Problem is supposed to define data structures meeting the following requirements:

- *Extensibility in both dimensions:* Both new data variants and new operations can be added to the existing framework. Moreover, existing operations are extensible to further support those new variants.
- *Strong static type safety:* The validity of interactions between data variants and operations is ensured by static type checks.
- *No modification or duplication:* The code is highly reusable. Existing code cannot be modified or duplicated.
- *Separate compilation and type-checking:* New extensions do not rely on recompilation of existing code. Static compilation can be done in a separate way.

Later Odersky and Zenger [2005a] added an additional requirement:

- *Independent extensibility:* It is expected that independently developed extensions are composable, so that they can be used jointly.

There have been many solutions to this problem [Carette et al., 2007; Odersky and Zenger, 2005a; Oliveira and Cook, 2012a; Swierstra, 2008; Torgersen, 2004; Wang and Oliveira, 2016] in various programming paradigms. Those techniques vary in multiple dimensions, and more interestingly, such a distinction reveals the trade-off between modularity and other aspects, and guides our concerns about modularization in new situations.

1.2 Classification of Modularization Techniques

Broadly speaking, there are two common research methodologies for modularization: proposing new languages (or language features), and correspondingly designing compilation rules for them; or capturing design patterns as reusable components or libraries in existing languages. The history has witnessed that both approaches stimulate each other to breakthrough bottlenecks for modularity. People design complicated patterns to modularize certain functionality of programs, by writing large amounts of code to fit in the current system. Later to simplify the code, those patterns are abstracted into new concepts and inspire novel language features or even paradigms.

From a different perspective, existing techniques can also be divided into *syntactic* and *semantic* modularization techniques. Some syntactic modularization approaches employ textual composition

techniques such as *superimposition* [Apel and Kästner, 2009]. Others may even build an additional layer on top of a target language with well-designed specifications; those specifications allow syntactic modularity, and a subsequent meta-programming procedure generates code in the target language for the whole. These syntactic techniques are quite popular in practice, due to their simplicity of implementation and use. Examples include many tools for developing *feature-oriented Software-Product Lines* (SPLs) [Apel and Kästner, 2009; Kästner et al., 2011], some *language workbenches* [Erdweg et al., 2015], as well as extensible parser generators [Gouseti et al., 2014; Grimm, 2006; Parr and Quong, 1995; Schwerdfeger and Van Wyk, 2009a; Viera et al., 2012; Warth et al., 2016a]. Yet as Kästner et al. [2011] note, a typical drawback of feature-oriented SPL implementations, which more generally applies to syntactic modularity approaches, is that such *"implementation mechanisms lack proper interfaces and support neither modular type checking nor separate compilation"*. Accordingly, the entire system becomes fragile and error-prone: a slight change at the front side can badly affect the generated programs.

Semantic modularization techniques go one step further in terms of modularity. They enable components or features to be modularly type-checked and separately compiled, which meets the desire of software engineers. Modular type-checking reports errors earlier in terms of the modular code people have written in the first place, thus enhancing robustness. Separate compilation avoids costly global compilation on every single change, which makes programming and debugging considerably efficient. Furthermore, semantic modularization enables the composition of compiled binaries as well as ensuring the type-safety of the code composed of multiple components, also modularizing and simplifying program reasoning. Examples include various approaches to *family polymorphism* [Ernst, 2001], *virtual classes* [Ernst et al., 2006] and so on.

Specifically, having realized the importance of separate compilation and modular type-checking, this dissertation employs semantic modularization techniques for three concrete problems (to introduce in Section 1.5) in existing programming languages, and hopefully motivate new language features in the future.

1.3 Is Modularity Everything?

Modularization techniques for an initial architecture of software design should (arguably) take the following aspects into consideration:

- *Interchangeability*: The modular components have the flexibility to be individually replaced with their alternatives, without affecting other modules.
- *Type-safety*: The modularization patterns are well encoded in a statically typed language. Separate compilation and modular type-checking are desired.
- *Conciseness*: Modules are nicely encapsulated and connected in concise glue code. Or rather, the underlying mechanism should be revealed as little as possible to clients.
- *Expressiveness*: The abstractions are expressive enough to deal with practical problems.
- *Efficiency*: Performance penalty from modularization is reasonable.

The first two are considered essential requirements (in semantic modularization), as covered by the Expression Problem. Nevertheless, existing techniques have reached different degrees regarding the remaining three properties. The conciseness, or simplicity issue is common in modularization approaches, for modularization usually integrates a couple of layers of abstractions, and for ensuring type safety they introduce considerable complexity to the code. Examples include the boilerplate composition code for data variants from different families in [Odersky and Zenger, 2005a], and the heavy use of *F-bounds* [Canning et al., 1989] in [Torgersen, 2004]. To avoid boilerplate, people utilize meta-programming techniques to generate code from templates, and further it spurs some extensions to existing languages.

There is a trade-off between expressiveness and predictability from the use of abstraction for modularization. More expressiveness basically indicates that the abstraction enlarges the domain of solvable problems. For instance, generalized Object Algebras [Oliveira et al., 2013; Zhang and Oliveira, 2017] are more expressive than Object Algebras [Oliveira and Cook, 2012a] in terms of explicit traversal control. And there are basic recursion patterns from category theory [Bird and de Moor, 1997; Herrlich and Strecker, 1973], including *catamorphism* and *anamorphism*; they are later generalized to *paramorphism* [Meertens, 1992], *apomorphism* [Vene and Uustalu, 1998] and even monadic variants [Fokkinga, 1994; Pardo, 1998], etc. Nonetheless, an abstraction with more genericity has fewer properties to capture, and thus makes programs harder to predict or reason about (one may think of "*entropy*" in information theory), and further makes it difficult to do optimization such as *deforestation* [Gill et al., 1993; Wadler, 1988].

The priority of efficiency somehow polarizes in industry and research. Industrial programs and software take the performance of great importance (especially on big data), so as to improve user experience, until developers realize that the long-term maintenance of non-modular code becomes more and more costly. However, the efficiency of a program can arguably be attributed to the infrastructure of the project and the implementation of real functionality separately. Hence one has to argue that his modularization techniques merely cause reasonable penalty on the efficiency. In some research areas, the importance of modularity is over-exaggerated without proper experiments and measurements on the execution time.

1.4 Modularity Issues of Today

Clearly the Expression Problem is not the destination, but a milestone in the history of modular programming. Existing approaches have contributed a lot to modularizing both data variants and operations simultaneously, but there are still two remaining issues:

- *The term "operation" is not identified clearly.* In most related literature, it only refers to *consumer* operations: operations that traverse a data structure, collect and integrate information. Oppositely, there are *producer* operations that build data structures, unfortunately they have not received much attention as they deserve. Note that a third category of operations, called *transformations*, can be encoded into consumers or producers, based on the actual implementations.

- *Modularization brings considerable boilerplate code in existing languages. Boilerplate* [Lammel and Jones, 2003] refers to verbose code that has to be repeated in many places with little or no

alteration. It is prevalent in design patterns, yet brings much inconvenience to programmers because of duplication.

This dissertation aims at *modularizing consumers and producers with type-safety, reusability and conciseness, without breaking the modularity of data variants*. Specifically, this dissertation studies three concrete research problems listed below:

I. Boilerplate traversals. Various language processing tools or libraries for programming languages, domain-specific languages, mark-up languages like HTML, or data-interchange languages like XML or JSON require complex Abstract Syntax Tree (AST) structures. Traversing those complex ASTs typically requires large amounts of tedious boilerplate code. For many operations most of the code simply walks the structure, and only a small portion of "interesting" code implements the functionality that motivated the traversal in the first place; such operations are called *structure-shy* [Lieberherr, 1996]. Having to explicitly write the entire traversal code is both tedious and error-prone.

Take a simple AST example from [Lammel and Jones, 2003] that describe a company infrastructure, presented by Figure 1.1 in Java code. Note that constructors are omitted for simplicity. A company consists of a number of departments, while each department has a manager that takes charge of a list of subunits. A unit can be either a single employee, or a department. Employees (including managers) are all persons with name and address information, together with their salaries. In the meantime, Figure 1.1 implements a consumer operation, which prints all employee names in a company, by realizing `printNames()` methods for all the concrete classes.

Right at this moment, one may start to realize that some code in `printNames()` could be boilerplate. All those code simply delegates the task of traversal to their sub-nodes, except `Person` which does the real interesting part: printing. This kind of structure-shyness forces programmers to duplicate a lot of boilerplate traversal code. Think about another operation that collects the salary bill of a company; most of the code will still do traversals tediously, except `Exployee` with its `salary`. Additionally, another drawback of this programming style is that it does not really support extensibility on operations, because new operations require modification on existing code across ASTs, violating the requirements in the Expression Problem.

Although existing approaches including *Adaptive Object-Oriented Programming* [Lieberherr, 1996] and *Strategic Programming* [Borovanskỳ et al., 1996; Visser and Benaissa, 1998a] aimed partly at reducing such boilerplate, they mostly adopt meta-programming techniques, hence sacrificing other desirable properties, including type-safety, separate compilation and even the extensibility specified by the Expression Problem. In object-oriented programming, a standard way is to write *default visitors* [Nordberg III, 1996] that capture certain traversal patterns. For this, one may define a `Visitor` interface with several `visit-` methods, and correspondingly inject `accept()` methods in ASTs. For example, a `visitDept()` method may have the following default implementation:

```
public void visitDept(Dept d) {
   d.manager.accept(this);
   for (SubUnit u : d.units) u.accept(this);
}
```

And this is still unsatisfactory: the Visitor pattern makes operations extensible, but simultaneously breaks the extensibility of data variants. Furthermore, default visitors have to be manually written.

```
class Company {
   List<Dept> depts;
   void printNames() {
      for (Dept d : depts)
         d.printNames();
   }
}

class Dept {
   String name;
   Employee manager;
   List<SubUnit> units;
   void printNames() {
      manager.printNames();
      for (SubUnit u : units)
         u.printNames();
   }
}

abstract class SubUnit {
   abstract void printNames();
}
class PUnit extends SubUnit {
   Employee employee;
   void printNames() { employee.printNames(); }
}
class DUnit extends SubUnit {
   Dept dept;
   void printNames() { dept.printNames(); }
}

class Employee {
   Person person;
   float salary;
   void printNames() { person.printNames(); }
}

class Person {
   String name, address;
   void printNames() { System.out.println(name); }
}
```

Figure 1.1: Printing the names of employees in a company.

[Lammel and Jones, 2003] provides simple generic traversal combinators in functional programming, but the efficiency is affected significantly by the heave use of run-time casts, and the approach still does not support extensibility on both data variants and operations.

II. Type-safe modular parsing. While modularization of consumers has received considerable attention among researchers, there is almost no related work on modularizing producers, namely the operations that build extensible ASTs, of which parsing is a typical representative. Existing approaches [Gouseti et al., 2014; Grimm, 2006; Parr and Quong, 1995; Schwerdfeger and Van Wyk, 2009a; Viera et al., 2012; Warth et al., 2016a] mostly support syntactic modularity, by allowing users to write separate grammars/specifications, and later they are combined and processed by *parser generators* as a whole. Consequently, such syntactic techniques do not support separate compilation or modular type-checking; the implementation becomes error-prone, and even a minor modification in the grammar requires a whole time-consuming recompilation.

In contrast with syntactic modularization, semantic modularization is more desired, for example when implementing the interpreters in the famous book *Types and Programming Languages* (TAPL) [Pierce, 2002], where languages are built and composed from reusable components. Modularizing parsers is not as easy as stacking bricks; additional challenges are introduced, when it involves left-recursive grammars, backtracking and the order of composition.

III. Modular unfolds. The modularization of consumers and producers has a different manifestation in functional programming. Existing work, especially *Algebra of Programming* [Bird and de Moor, 1997] and *Data types à la carte* [Swierstra, 2008] that have theoretical foundations in category theory [Herrlich and Strecker, 1973], have demonstrated that data operations can be realized by using recursion schemes. Specifically, consumers correspond to *f-algebras* and *catamorphisms* (the generic fold), and dually producers correspond to *f-coalgebras* and *anamorphisms* (generic unfolds). Unfortunately, although the work indicates that modularizing consumers comes from the composability of `f`-algebras (which further implies a solution to the Expression Problem), modularizing producers by composing `f`-coalgebras has been much less explored despite the duality. Additionally, there are few applications in this area. We expect to modularize the construction of data structures into reusable components, and provide producer/coalgebra combinators to compose them in a convenient and type-safe way without boilerplate code.

1.5 Contributions

To tackle the aforementioned research problems, this dissertation proposes a couple of semantic modularization approaches and libraries, together with a few case studies to demonstrate the practicality. More aspects in Section 1.3 have been taken into concern. More precisely, contributions of this dissertation for the three research topics are:

I. Boilerplate traversals. This dissertation presents a Java framework called *Shy* that allows users to define type-safe and extensible structure-shy consumers. *Shy* uses *Object Algebras* [Oliveira and Cook, 2012a], a solution to the Expression Problem, as the underlying mechanism to describe ASTs and to write operations over ASTs for traversals. As a result, both data variants and operations are extensible in a semantically modular way. *Shy* captures four different types of consumers: *queries*,

transformations, *generalized queries* and *contextual transformations*. Those patterns are abstracted into templates, and are expressive enough to address many practical problems. Furthermore, *Shy* utilizes Java annotations to generate generic traversal code for those patterns; meta-programming in *Shy* is merely for auxiliary use. Consequently, the amount of code that users need to write is significantly smaller. The case study on a domain-specific language shows that for a large number of traversals, only 4% to 21% of the AST cases had to be implemented in comparison with code written without *Shy*, while the performance is comparable to the vanilla implementation.

II. Type-safe modular parsing. This dissertation investigates the algorithmic challenges of type-safe modular parsing, including *left-recursion*, *manual backtracking*, and *global ordering of composition*. We also investigate the challenges in terms of extensibility and reusability. Finally, a technique for writing type-safe modular parsers is presented, by integrating *Packrat parsing* [Ford, 2002], multiple inheritance and Object Algebras [Oliveira and Cook, 2012a] together in Scala code. Consequently the methodology allows multiple dimensions of extensibility, and implies a general approach for modularizing producers with Object Algebras. The evaluation is conducted on a case study with parsers of 18 TAPL interpreters, showing its reasonable performance and 69% reduction of code size in total.

III. Modular unfolds. This dissertation investigates techniques for composing and modularizing `f`-coalgebras. We show that the natural and generic composition of `f`-coalgebras leads to products (instead of sums) at the top-level nodes, which further gives *product forests* from the generic unfold. We then show that the more familiar, and expected *sums-of-product* structure can be built as a second step after the construction of a product forest. In order to eliminate the construction of the intermediate product forest, and to directly build a sums-of-product structure, we show that specialized composition combinators can be derived for coalgebras, and we further generalize this deforestation to monadic variants. Such a methodology motivates the development of *SCCL*, a library of specialized coalgebra combinators for various composition strategies. To demonstrate the practicality of *SCCL*, modular random generators, enumerators, and modular small-step semantics are encoded by modularizing `f`-coalgebras. Finally, a case study on random generators and enumerators conducts our evaluation.

Summary To summarize, the contributions of this dissertation are listed as follows, of which some are labelled with "Part I/II/III" for a correspondence to the three concrete problems:

- Part I: *Design patterns for generic traversals.* We provide a set of design patterns for various types of traversals (consumers) using Object Algebras, including: *queries*, *transformations*, *generalized queries* and *contextual transformations*. Both data variants and those patterns support semantic extensibility.
- Part I: *The Shy Java framework.* We show that those generic traversal patterns can be automatically generated for a given Object Algebra interface. The *Shy* framework realizes this idea by using Java annotations to automatically generate generic traversals.
- Part II: *A technique for modular parsing.* We present a technique that allows the development of semantically modular parsers. The technique relies on the combination of Packrat parsing, multiple inheritance and Object Algebras, presented in Scala code.

- PART II: *A methodology for writing modular parsers.* We identify possible pitfalls using parser combinators. To avoid such pitfalls, we propose guidelines for writing parsing code using *left-recursion* and *longest-match composition*.

- PART III: *Product forests.* We identify product forests as the natural result arising from unfolding `f`-coalgebras composed with the general coalgebra combinator for *product* types, and show how to recover a fixpoint of sums-of-products from a product forest in a subsequent step.

- PART III: *Techniques for deforestation of product forests.* We show a methodology for eliminating product forests and directly obtaining a fixpoint of sums-of-products, by replacing the generic `f`-coalgebra operator for product types by more specialized composition operators for co-products.

- PART III: *Fusion and equivalence theorems.* We identify several useful fusion theorems to show the correctness of deforestation techniques. We also present and prove theorems that show equivalent ways to transform product forests into fixpoints of sums-of-products.

- PART III: *Monadic variants.* We also discuss monadic variants of our techniques together with the corresponding fusion theorems. The monadic variants are interesting because they enable more interesting selection functions.

- PART III: *The SCCL library.* We propose *SCCL* as a library of specialized coalgebra combinators, with several useful composition strategies for modularizing producers. They are developed in Haskell, with the use of *monad transformers* for extensibility over functionality.

- *Applications and examples.* For modular generic traversals (consumers), We show various examples such as free variables and substitution operations, implemented concisely with *Shy*. Moreover we illustrate how certain kinds of desugarings can be implemented using transformations, and how multiple transformations can be chained together in a *pipeline*. For modular producers, we show three applications that can be achieved by modularizing `f`-coalgebras: *modular random generators and enumerators*, *modular small-step semantics*, and *modular monadic parsing*. They require various composition strategies captured by our *SCCL* library. The resulting code is well encapsulated and concise to users.

- *Case studies.* For all three parts, case studies are conducted to illustrate modularity as well as other concerns, including conciseness, expressiveness, and efficiency. For Part I, we illustrate the benefits of *Shy* using a case study based on the QL domain-specific language. The results of our case study show significant savings in terms of user-defined traversal code. The case study also shows that *Shy* does not incur significant performance overhead compared to a regular AST-based implementation. For Part II, we conduct a case study with 18 interpreters from the TAPL book. The case study shows the effectiveness of modular parsing in terms of reuse, resulting in around 69% of code size reduction. For Part III, we model a set of random generators and enumerators for modular interpreters. In the case study, the generators are modelled using monadic `f`-coalgebras, and composed under various strategies, showing great flexibility and reusability.

- *Investigation in different paradigms.* We study the modularization of consumers and producers, in both object-oriented programming and functional programming (Java, Scala, and Haskell). Our discussion tries to explore the isomorphism between patterns implemented in different paradigms, and connect the techniques in theoretical aspects.

All the source code is available online[1].

1.6 Organization

This dissertation is generally organized as follows:

- Chapter 2 contains some necessary preliminaries to make the dissertation self-contained.
- Chapter 3 presents the *Shy* Java framework that abstracts consumers by capturing generic traversal patterns in Object Algebras, and generates templates for those patterns with annotation processing, hence removing much boilerplate code and meanwhile maintaining semantic modularity. The case study on the domain-specific language QL for questionnaires is also included.
- Chapter 4 presents the methodology for writing semantically modular parsers in Scala, by integrating Packrat parsing, multiple inheritance and Object Algebras. Additionally, the case study on parsing TAPL interpreters is included.
- Chapter 5 presents the methodology for modularizing producers (unfolds) by modularizing `f`-coalgebras, and the *SCCL* library that contains several specialized coalgebra combinators.
- Chapter 6 presents the case study on modularizing random generators and enumerators using *SCCL*.
- Chapter 7 reviews some related work on the areas we have touched.
- Chapter 8 concludes and discusses possible directions for future work.

The main body of this dissertation is partly based on two publications by the author and in collaboration with others. Specifically:

- Chapter 3: "Scrap your Boilerplate with Object Algebras". Haoyuan Zhang, Zewei Chu, Bruno C. d. S. Oliveira and Tijs van der Storm. In *Proceedings of the 30th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2015)*.
- Chapter 4: "Type-Safe Modular Parsing". Haoyuan Zhang, Huang Li and Bruno C. d. S. Oliveira. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*.

while Chapter 5 is unpublished joint work with Bruno C. d. S. Oliveira and Shin-Cheng Mu.

[1] Part I: `https://github.com/ZeweiChu/ObjectAlgebraFramework`
Part II: `https://github.com/hy-zhang/parser`
Part III: `https://github.com/hy-zhang/SCCL`

Chapter 2

Background

This chapter sets the stage for our studies in designing flexible and reusable programs, and present prerequisite background for understanding this dissertation. Section 2.1 introduces the "Algebra of Programming" pattern in Haskell. Following the detailed description of the Expression Problem (EP) in Section 1.1, Section 2.2 and 2.3 discuss a few existing approaches/attempts for EP in Java (representative for object-oriented programming) and Haskell (representative for functional programming), respectively. Section 2.4 briefly introduces two meta-programming techniques in Java: annotation processing and reflection. Section 2.5 introduces the Scala Packrat Parsing as a primary technique used in Chapter 4. Section 2.6 briefly mentions monad transformers in Haskell, which is a prerequisite for Chapter 5. Finally Section 2.7 presents the usage of QuickCheck for defining generators and testing programs.

2.1 Algebra of Programming

There has been well-developed work on category theory [Herrlich and Strecker, 1973] and the "Algebra of Programming" (AoP) [Bird and de Moor, 1997] for categorical representations of datatypes. The Data Types à la Carte approach by [Swierstra, 2008] helped popularising various AoP concepts in practice. This part illustrates some essential concepts and their implementations in the context of Haskell.

FUNCTOR Theoretically, a *functor* is a homomorphism between categories [Bird and de Moor, 1997]. Haskell uses the `Functor` class to represent all endofunctors on its own category *Hask*, with kind `* -> *`. A `Functor` requires an implementation of the generic mapping function:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Functors can be composed flexibly in various ways. Besides the nested composition, products and coproducts are also commonly used in Haskell:

```
data (f • g) a = Comp (f (g a))
data (f ⊗ g) a = Prod (f a) (g a)
data (f ⊕ g) a = Inl (f a) | Inr (g a)
```

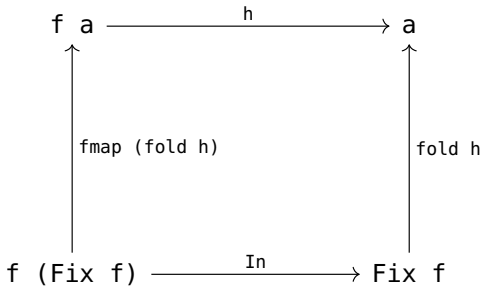

Figure 2.1: Catamorphism.

```
fold :: Functor f => (f a -> a) -> Fix f -> a
fold h = h . fmap (fold h) . out
```

Figure 2.2: The generic fold.

Furthermore, taking the *fixpoint* of a functor represents inductive datatypes in AoP. The fixpoint has the following definition:

```
data Fix f = In { out :: f (Fix f) }
```

where `In` and `out` witness the isomorphism `Fix f` $\approx$ `f (Fix f)`; they correspond to the *initial algebra* and the *final coalgebra* respectively in category theory.

ALGEBRA AND CATAMORPHISM Consumers, namely operations that consume data structures are represented by *algebras*. An `f`-algebra has the following type:

```
type Alg f a = f a -> a
```

where `a` is called the *carrier*. By the type, an algebra only conducts a one-level calculation.

From category theory, there is a unique homomorphism from the initial algebra `(Fix f, In)` to an arbitrary algebra `(a, h)`, where `h` has type `Alg f a`. Such a homomorphism indicates the *catamorphism* (also known as the *fold*) shown in Figure 2.1. Catamorphism is a generic recursion pattern that "lifts" a one-level algebra to a consumer that can readily take a `Fix f` structure, exhaustively fold/collapse the structure and end up with a single result. It represents a bottom-up traversal strategy.

Figure 2.1 immediately derives the implementation of `fold` in Figure 2.2, since `In` and `out` are inverse operations. Firstly, the operator `out` unpacks it to `f (Fix f)`, before recursive calls in `fmap (fold f)` turn it into an `f a` structure. Finally, the algebra `h` is applied to yield the result of type `a`. This combinator also appears as the banana brackets "$(\!|\,.\,|\!)$" in some literature.

COALGEBRA AND ANAMORPHISM Oppositely to consumers, there are producers that *build* data structures using certain recursion patterns. Coalgebras and the anamorphism elegantly dualize algebras and catamorphism, respectively. A *coalgebra* takes a carrier `a`, and returns an `f a` value:

```
type CoAlg f a = a -> f a
```

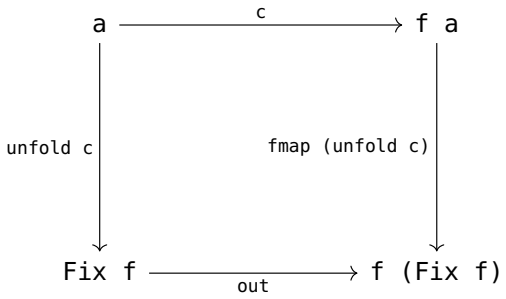

Figure 2.3: Anamorphsim.

```
unfold :: Functor f => (a -> f a) -> a -> Fix f
unfold c = In . fmap (unfold c) . c
```

Figure 2.4: The generic unfold.

Similarly, the homomorphism from an arbitrary coalgebra `c :: a -> f a`, to the final coalgebra `out` is called an *anamorphism* (or an *unfold*). Figure 2.3 shows the diagram of convertions between those types. It observes that the unfold generates a `Fix`-value from an input `a`. Following the other path, the implementation of such a generic unfold can be realized by Figure 2.4. The coalgebra is first applied to a seed of type `a`, generating an `f`-structure of `a`'s. Then they are recursively used to build subtrees, before wrapped up by constructor `In` in the end. Some related literature uses a simpler notation "$[\![.]\!]$". Anamorphism represents a basic top-down strategy of structure creation.

More Generalized Recursion Schemes Recursion schemes capture specific patterns of recursion to process data structures. Programs that use recursion schemes can be written, transformed, optimized and reasoned about in principled and concise ways. Besides the aforementioned catamorphism and anamorphism, there has already been well-developed research on exploring more generalized and expressive recursion schemes.

Worthy to be mentioned in the family of recursion schemes are the following ones. *Hylomorphism* is simply the combination of a catamorphism and an anamorphism:

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo c h =  h . fmap (hylo c h) . c
```

An equivalent definition composes `fold h` to `unfold c`. Besides, *paramorphism*, first proposed by [Meertens, 1992], is a generalization of catamorphism, so that it can track the data substructures on which recursive calls are applied. A Haskell implementation is given below:

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para h = h . fmap (\n -> (n, para h n)) . out
```

Similarly, Vene and Uustalu [1998] proposed *apomorphism* as a generalization of anamorphism, where the coalgebra can either generate new seeds or substructures (`Fix f` values) directly:

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo c = In . fmap c' . c
```

```
  where c' e = case e of
    Left l    -> l
    Right r -> apo c r
```

On the other hand, Fokkinga [1994] and Pardo [1998] generalized catamorphism and anamorphism with side effects. In Haskell, they lead to the monadic fold and unfold as follows, using `Traversable` [Gibbons and Oliveira, 2009; Mcbride and Paterson, 2008] instances:

```
type AlgM m f a = f a -> m a

foldM :: (Traversable f, Monad m) => AlgM m f a -> Fix f -> m a
foldM f = f <=< mapM (foldM f) . out

type CoAlgM m f a = a -> m (f a)

unfoldM :: (Monad m, Traversable f) => CoAlgM m f a -> a -> m (Fix f)
unfoldM h = fmap In . (mapM (unfoldM h) <=< h)
```

These variations are able to model more complicated calculations. Nevertheless, more powerfulness implies more generality, and hence implies less abstraction with fewer properties. There is always a trade-off between abstraction and expressiveness.

2.2 Solutions to EP in Java

The Expression Problem (EP), firstly coined by Wadler [1998] for describing the requirements in modularizing both data variants and operations, has been introduced in Section 1.1. Researchers from the object-oriented area have proposed many solutions to the Expression Problem. In this section, we choose Java as the language for our illustration. The following techniques are mostly not restricted to Java though.

Design Goal We anticipate to model a simple arithmetic expression language in Java by embedding. The initial configuration is comprised of two data variants: literals and additions, together with an operation that evaluates an expression to a number. Later on, such a language may be extended with multiplications, and an additional behavior: pretty-printing.

2.2.1 A Non-Solution: The Interpreter Pattern

The Interpreter pattern, one of the twenty-three well-known GoF design patterns [Gamma, 1995], is perhaps a most commonly used approach for abstracting tree structures. It is effectively a traditional class-based approach, with expressiveness and simplicity of implementation using inheritance.

Figure 2.5 (left) gives an implementation of the expression language. `Exp` defines an interface to integrate valid operations on the language. Here `eval()` represents the evaluation. Afterwards, the abstract syntax tree (AST) is described by a number of classes implementing that interface. Note that the Composite pattern is often used to encode recursive structures. Each class corresponds to a data variant, with the members, the constructor as well as a concrete implementation of `eval()` by method overriding.

```
interface Exp {
   public int eval();
}

class Lit implements Exp {
   int i;
   public Lit(int i) { this.i = i; }
   public int eval() {
      return i;
   }
}

class Add implements Exp {
   Exp e1, e2;
   public Add(Exp e1, Exp e2) {
      this.e1 = e1;
      this.e2 = e2;
   }
   public int eval() {
      return e1.eval() + e2.eval();
   }
}
```

```
class Mul implements Exp {
   Exp e1, e2;
   public Mul(Exp e1, Exp e2) {
      this.e1 = e1;
      this.e2 = e2;
   }
   public int eval() {
      return e1.eval() * e2.eval();
   }
}
```

Figure 2.5: The INTERPRETER pattern (left). Multiplication as an extension (right).

EXTENSION: NEW VARIANT It is observed that `Lit` and `Add` can be developed independently, thus the extensibility of data variants can be achieved by adding a new class. Figure 2.5 (right) shows multiplication as an extension.

EXTENSION: NEW OPERATION Unfortunately, traditional OO does not have natural extensibility on operations. To allow pretty-printing of expressions, inserting an additional abstract method into `Exp` breaks the third requirement of EP, namely "no modification on existing code". Not doing so but extending `Exp` with inheritance is still problematic, as shown below:

```
interface ExtExp extends Exp {
   String print();
}

class ExtAdd extends Add implements ExtExp {
   public ExtAdd(Exp e1, Exp e2) {...}
   public String print() {
      return "(" + e1.print() + " + " + e2.print() + ")";
         // does not type-check!
   }
}
```

The above code fails on type checking, since `e1` and `e2` belong to the old expression type. It is worth mentioning that *type-refinement* can possibly be applied to those members in Java or Scala [Wang and Oliveira, 2016], but the authors consider that approach to be restricted when *binary* and *producer methods* are introduced. In short, the traditional OO approach merely realizes partial extensibility. Interestingly, it reveals the key to the Expression Problem, namely how to untangle the design of data representations from operations.

2.2.2 The Opposite Side: The Visitor Pattern

Compared with functional programming where pattern-matching is a frequently used technique, checking class membership is considered a violation of object-oriented programming. Nevertheless, the Visitor pattern [Gamma, 1995] simulates this functional decomposition style. It realizes a separation of designing operations from the abstract syntax structure. Yet this is still a non-solution to EP; it only flips the coin to the opposite side, namely adding operations is easier while adding variants becomes difficult.

There are various ways to classify visitors: *imperative* versus *functional*; *internal* versus *external*; etc. [Oliveira, 2007; Oliveira et al., 2008]. Figure 2.6 implements internal functional visitors for a representative. The `Visitor` interface is defined with respect to the shape of the abstract syntax, where each method is specific to a data variant. It is parametrized to abstract the operations applicable to expressions, where `E` denotes the return type of an operation to be instantiated.

In the definition of `Exp`, with genericity, the `accept()` method is the entrance to execute a visitor for a traversal. Since it is an *internal* visitor, generic traversal code is encapsulated inside the classes `Lit` and `Add`, by overriding `accept()`, threading the visitor on recursive structures, and calling the corresponding visit method at the top. Consequently, the pattern conducts a bottom-up traversal strategy.

Finally, as a concrete visitor, `Eval` instantiates the type parameter of `Visitor` into `Integer`, and provides implementations for evaluating literals and additions.

Extension: New Operation Similarly to `Eval`, a different instantiation can be modularly added to the existing framework. Figure 2.7 presents a pretty-printing behavior.

Extension: New Variant Unfortunately, the traditional Visitor pattern sacrifices the extensibility on data variants. To extend the current abstract syntax, it is necessary to substitute `Visitor` with a subtype that can handle new variants. Yet `Visitor` appears at a contravariant parameter position in `Exp`, further requiring programmers to modify or duplicate the code where `Exp` is referred. In a similar way to the Interpreter pattern, it merely realizes partial extensibility.

2.2.3 Object Algebras

Object Algebras are an actual solution to the Expression Problem, proposed by Oliveira and Cook [2012a]. The original pattern of Object Algebras is strongly connected to internal visitors, *Church encodings* [Church, 1936], and folds in functional programming. The relationship between internal visitors and Church encodings is discussed by [Buchlovsky and Thielecke, 2006], furthermore, a generalized version of Object Algebras [Oliveira et al., 2013] can model external visitors, but this is beyond the preliminaries of the thesis.

```
interface Visitor<E> {
   E visitLit(int i);
   E visitAdd(E e1, E e2);
}

interface Exp {
   <E> E accept(Visitor<E> v);
}

class Lit implements Exp {
   int i;
   public Lit(int i) { this.i = i; }
   public <E> E accept(Visitor<E> v) {
      return v.visitLit(i);
   }
}

class Add implements Exp {
   Exp e1, e2;
   public Add(Exp e1, Exp e2) {
      this.e1 = e1;
      this.e2 = e2;
   }
   public <E> E accept(Visitor<E> v) {
      return v.visitAdd(e1.accept(v), e2.accept(v));
   }
}

class Eval implements Visitor<Integer> {
   public Integer visitLit(int i) {
      return i;
   }
   public Integer visitAdd(Integer e1, Integer e2) {
      return e1 + e2;
   }
}
```

Figure 2.6: The *Visitor* pattern.

Object Algebra Interface An *Object Algebra interface* representing expressions is isomorphic to `Visitor` in Figure 2.6. For naming conventions, the interface is usually postfixed with `"Alg"`:

```
interface ExpAlg<E> {
   E lit(int i);
   E add(E e1, E e2);
```

```
class Print implements Visitor<String> {
   public String visitLit(int i) {
      return "" + i;
   }
   public String visitAdd(String e1, String e2) {
      return "(" + e1 + " + " + e2 + ")";
   }
}
```

Figure 2.7: Pretty-printing as a visitor.

```
   }
```

where `E` is often called a *carrier*, for capturing recursive structures.

One key difference between Object Algebras and the VISITOR pattern, is that `Exp` and its `accept()` method are no longer required, unless for encapsulation. Thus Object Algebra interfaces are responsible to describe the shape of data variants; in fact, they have a strong connection to ABSTRACT FACTORY [Gamma, 1995] interfaces. The difference is that, for the VISITOR pattern, `Lit` and `Add` in Figure 2.6 will be defined as factory methods instead, with return type `Exp`; in Object Algebra interfaces, however, the return type is generic. Later an Object Algebra interface will be instantiated into a concrete factory, called an *Object Algebra*.

Rather than building explicit `Exp` objects, Object Algebras provide "recipes" for creating values [Oliveira and Cook, 2012a], based on Church encodings. As a concrete example, the following code defines a polymorphic function standing for `"2 + 3"`, ready to accept any concrete factory and produce results accordingly:

```
<E> E build(ExpAlg<E> alg) {
   return alg.add(alg.lit(2), alg.lit(3));
}
```

OBJECT ALGEBRAS Object Algebras encode the behaviors on data structures. An *Object Algebra* instantiates the abstract carriers to certain types, and correspondingly provides implementations for all variants. As a result, implementing an Object Algebra is no different from implementing a visitor. For instance, the evaluation algebra is equivalent to the `Eval` in Figure 2.6:

```
class Eval implements ExpAlg<Integer> {
   public Integer lit(int i) {
      return i;
   }
   public Integer add(Integer e1, Integer e2) {
      return e1 + e2;
   }
}
```

To apply an evaluation, such a factory is fed to the above `build` function:

```
Integer x = build(new Eval());
```

yielding `5` as expected.

Extension: New Variant An Object Algebra interface can easily be extended with new data variants by inheritance. Theoretically, data extension as a sum of types, manifests as the product in Church-encoded functions, and OO inheritance can be used to model such products [Buchlovsky and Thielecke, 2006; Oliveira and Cook, 2012a]. The following code extends the expression language with multiplication:

```
interface MulAlg<E> extends ExpAlg<E> {
   E mul(E e1, E e2);
}
```

And correspondingly, an abstract factory taking `MulAlg<E>` as a parameter can use `mul` for constructing multiplications.

Extension: New Operation Object Algebras allow both dimensions of extensibility. Adding a new operation is as easy as the *Visitor* pattern. Below code defines a pretty-printing algebra as in Figure 2.7:

```
class Print implements ExpAlg<String> {
   public String lit(int i) {
      return "" + i;
   }
   public String add(String e1, String e2) {
      return "(" + e1 + " + " + e2 + ")";
   }
}
```

Algebras are further reusable when the abstract syntax expands. It is straightforward to modularly add pretty-printing on multiplications:

```
class MulPrint extends Print implements MulAlg<String> {
   public String mul(String e1, String e2) {
      return "(" + e1 + " * " + e2 + ")";
   }
}
```

2.2.4 Other Approaches

There has been lots of other related work on finding solutions to the Expression Problem in object-oriented programming [Odersky and Zenger, 2005a; Torgersen, 2004; Wang and Oliveira, 2016]. Some of them have a requirement for advanced language features, including *multi-methods* [Chambers and Leavens, 1995], *open classes* [Clifton et al., 2000], *abstract type members* [Odersky and Zenger, 2005b], *F-bounded polymorphism* [Canning et al., 1989], and so on. While details of those approaches will not be covered in the thesis, we briefly review them in Chapter 7.

```
data Lit     = Lit Int
data Add a b = Add a b

class Expr x
instance Expr Lit
instance (Expr a, Expr b) => Expr (Add a b)

class Expr x => Eval x where
  eval :: x -> Int

instance Eval Lit where
  eval (Lit n) = n

instance (Eval a, Eval b) => Eval (Add a b) where
  eval (Add x y) = eval x + eval y
```

Figure 2.8: Encoding the expression language with algebraic datatypes and type classes.

2.3 Solutions to EP in Haskell

The problem of extensibility has also been heavily studied in functional programming. Below are a few representatives of the solutions using Haskell. It is interesting to observe the strong connection between object-oriented programming and functional programming; the same notion manifests differently in both contexts.

2.3.1 A Partial Solution: Polymorphic Datatypes and Type Classes

Before introducing the EP solutions more related to this dissertation, there is a folklore approach in Haskell which uses *algebraic data types* for variants, and *type classes* to process operations. Traditionally, the definition of datatypes is closed in Haskell; it has to be opened for extension by polymorphism. Besides, type classes allow an object-oriented style in functional programming, where functions are packed and dispatched with respect to types. The advantage of using type classes, compared to object-oriented programming, is that it separates operations from data definition.

Consider an encoding of the expression language in Figure 2.8. Literals and additions are defined by data constructors, respectively. `Add` takes two type parameters for its sub-terms, thereby an expression like `"3 + 5"` is expressed by `Add (Lit 3) (Lit 5)`. Furthermore, `Expr` adds a constraint on those data variants, so as to encode multiple different syntax in future extensions.

In this approach, valid data operations are captured by type classes. `Eval` packs an `eval` function for the evaluation. The semantics on literals and additions are provided in a modular style. Finally, running `"eval (Add (Lit 3) (Lit 5))"` in GHCi yields `5` immediately.

Extension: New Variant Multiplications and the corresponding semantics are free to be added to the current framework modularly. As follows:

```
data Mul a b = Mul a b
```

```
instance (Expr a, Expr b) => Expr (Mul a b)

instance (Eval a, Eval b) => Eval (Mul a b) where
  eval (Mul x y) = eval x * eval y
```

Extension: New Operation Pretty-printing is further applicable with another type class:

```
class Expr x => Print x where
  pprint :: x -> String

instance Print Lit where
  pprint (Lit n) = show n

instance (Print a, Print b) => Print (Add a b) where
  pprint (Add x y) = "(" ++ pprint x ++ " + " ++ pprint y ++ ")"

instance (Print a, Print b) => Print (Mul a b) where
  pprint (Mul x y) = "(" ++ pprint x ++ " * " ++ pprint y ++ ")"
```

Denote `e` for the expression `"Mul (Lit 2) (Add (Lit 3) (Lit 5))"`. Executing `eval` and `pprint` on `e` gives the desired results `16` and `"(2 * (3 + 5))"`, respectively. Nevertheless, this approach does not solve the Expression Problem totally. By checking the type of `e`, we obtain `Mul Lit (Add Lit Lit)`. It indicates that the type of an expression depends on its value; consequently, it is hard to unify different expressions with the same type, for example when defining a parse function for the syntax.

2.3.2 Finally Tagless

The idea originates in [Hinze, 2006] which represents different encodings of datatypes using Haskell type classes. Later Oliveira et al. [2006] applied some variations to the encodings and implied a solution to the Expression Problem, before *Finally tagless* (or *tagless final*) [Carette et al., 2007] actually popularized the technique in realizing interpretation of simple embedded DSLs in functional programming.

In this approach, an abstract syntax is Church-encoded into a type class, which is isomorphic to an Object Algebra interface; and operations are realized by instances of the type class. While the original paper gives a more complicated representation of interpreters, Figure 2.9 simplifies it a bit for expressions. The code clearly indicates the relationship between finally tagless and Object Algebras. The expression `"2 + 3"` is represented by an abstract Church-encoded value, with the constraint `Expr e`. Running `"eval twoPlusThree"` yields `5` correctly.

Extension: New Variant Haskell emulates OO inheritance between type classes, allowing an extension of multiplication:

```
class Expr e => MulExpr e where
  mul :: e -> e -> e
```

```
class Expr e where
  lit :: Int -> e
  add :: e -> e -> e

newtype Eval = Eval { eval :: Int }

instance Expr Eval where
  lit n   = Eval n
  add x y = Eval $ eval x + eval y

twoPlusThree :: Expr e => e
twoPlusThree = add (lit 2) (lit 3)
```

Figure 2.9: The expression language using a finally-tagless representation.

```
instance MulExpr Eval where
  mul x y = Eval $ eval x * eval y
```

EXTENSION: NEW OPERATION The extensibility on operations is straightforward by defining new instances of `Expr` and `MulExpr`:

```
newtype Print = Print { pprint :: String }

instance Expr Print where
  lit n   = Print $ show n
  add x y = Print $ "(" ++ pprint x ++ " + " ++ pprint y ++ ")"

instance MulExpr Print where
  mul x y = Print $ "(" ++ pprint x ++ " * " ++ pprint y ++ ")"
```

2.3.3 Data Types à la Carte

Finally tagless and Object Algebras realize both dimensions of extensibility in EP, at the cost of representing data in the form of functions based on Church encoding. Such a representation, however, can be *reified* into free structures in the *Data Types à la Carte* (DTC) approach. The relationship between both approaches has a correspondence to *shallow embeddings* versus *deep embeddings* [Gibbons and Wu, 2014].

Swierstra [2008] proposed DTC as another solution to EP in the context of Haskell. In fact, the approach is a realization of categorical concepts from Algebra of Programming [Bird and de Moor, 1997]. His work innovatively increases the practicability of AoP in Haskell.

REPRESENTATION DTC represents inductive data types as the fixpoint of functors, and certain operations by defining algebras. Firstly, a functor for encoding literals and additions can be defined as follows:

```
data ExpF e = Lit Int | Add e e     deriving Functor
```

Notice that it still looks similar to an Object Algebra interface, or a tagless final type class, whereas Haskell algebraic datatypes are used for reification of the structure.

As the second step, taking the fixpoint of `ExpF` gives the type of expression objects. Construct `"2 + 3"` for example:

```
e :: Fix ExpF
e = In $ Add (In $ Lit 2) (In $ Lit 3)
```

Algebra and Generic Fold Operations, more specifically consumers, are defined as algebras with the generic fold. For instance, the following type class defines the evaluation of expressions, where `evalAlgebra` instantiates the carrier to `Int`:

```
class Functor f => Eval f where
  evalAlgebra :: f Int -> Int
```

Then `eval` feeds the evaluation algebra to the generic `fold` from Figure 2.2:

```
eval :: Eval f => Fix f -> Int
eval = fold evalAlgebra
```

Now we instantiate `f` to `ExpF` in `Eval` and attach an implementation:

```
instance Eval ExpF where
  evalAlgebra (Lit n)   = n
  evalAlgebra (Add x y) = x + y
```

In that case, applying `eval` to `e` yields `5` instantly.

Extension: New Variant Multiplication is introduced by another functor, together with its evaluation algebra:

```
data MulF e = Mul e e      deriving Functor

instance Eval MulF where
  evalAlgebra (Mul x y) = x * y
```

In DTC, the key idea of composite structure is taking the *coproduct* (or *sum*) of functors:

```
data (f ⊕ g) e = Inl (f e) | Inr (g e) deriving Functor
```

where `Inl` and `Inr` are injections. A linearization of coproducts can be wrapped into a single fixpoint. Additionally, the following code realizes automatic composition of evaluation algebras for a coproduct:

```
instance (Eval f, Eval g) => Eval (f ⊕ g) where
  evalAlgebra (Inl e) = evalAlgebra e
  evalAlgebra (Inr e) = evalAlgebra e
```

Now `Fix (ExpF ⊕ MulF)` involves literals and addition, but also multiplication. Subsequently, `eval` is ready to consume `Fix (ExpF ⊕ MulF)` structures due to its genericity.

```
class Functor f => Print f where
  printAlgebra :: f String -> String

instance (Print f, Print g) => Print (f ⊕ g) where
  printAlgebra (Inl e) = printAlgebra e
  printAlgebra (Inr e) = printAlgebra e

instance Print ExpF where
  printAlgebra (Lit n)   = show n
  printAlgebra (Add x y) = "(" ++ x ++ " + " ++ y ++ ")"

instance Print MulF where
  printAlgebra (Mul x y) = "(" ++ x ++ " * " ++ y ++ ")"
```

Figure 2.10: Adding the pretty-printing algebra in DTC.

EXTENSION: NEW OPERATION Algebras can be independently defined for modularity. Figure 2.10 presents an implementation of pretty-printing. It demonstrates that DTC settles the Expression Problem. Importantly, one shall notice that the use of type classes in DTC is not as necessary as the traditional FP approach in Section 2.3.1. In the previous approach, type classes define operations as recursive functions; the recursion is opened and the computation is automatically conducted by instances. In DTC, they are merely used for the automatic composition of algebras; alternatively, it is sufficient to define and compose algebras using simple functions. The recursion pattern is captured by the generic `fold`.

2.4 Java Annotation Processing and Reflection

This section introduces some prerequisite knowledge about Java annotation processing and Java reflection. They are both commonly used meta-programming techniques, involved in some existing libraries including JUnit [Beck and Gamma, 1998], Lombok[1], Retrofit[2], etc. In Java, the APIs for both mechanisms are:

- *Annotation processing:* API located in `javax.annotation.processing` package. Annotations are defined in the form of "`@XXX`", implemented for certain behaviors, and processed during compilation. They can generate code into new source files, or even manipulate the abstract syntax tree before *.class* files are generated, using Lombok for example. The generated code will require another compilation process, hence type safety is guaranteed by the Java compiler.
- *Reflection:* API located in `java.lang.reflect` together with other related packages. Reflection allows programs to introspect and modify themselves at the runtime execution, on internal properties such as classes, members and methods dynamically. Compared with annotations, reflection can easily raise runtime errors.

[1] `https://projectlombok.org/`
[2] `https://square.github.io/retrofit/`

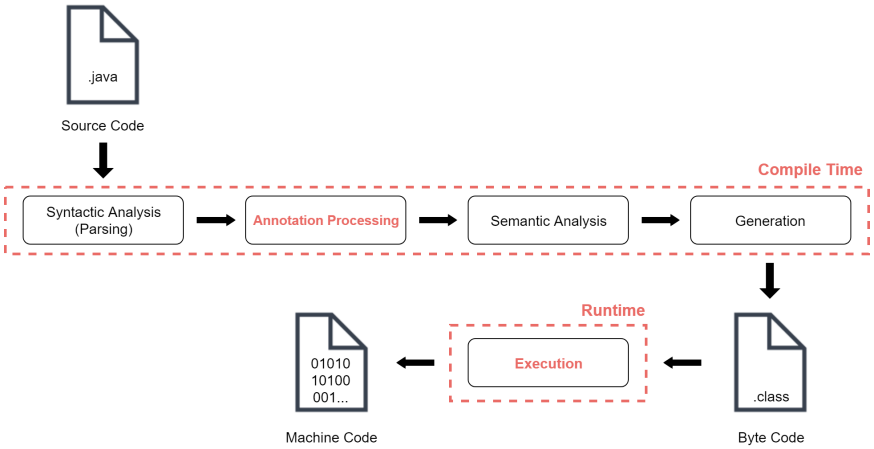

Figure 2.11: Compilation and execution process of Java programs. Annotations are handled in "Annotation Processing", while reflection takes place during "Execution".

Figure 2.11 illustrates where they take place in the compilation and execution of Java programs.

2.5 Scala Packrat Parsing

Parsing is a fundamental procedure in the design of compilers and interpreters, to connect abstract syntax with concrete syntax. A conventional approach for its implementation is to use *parser generators*, where the overall syntax is encoded into special grammar rules, before code generation produces the real parsing code automatically. On the other hand, *parser combinators*, originating in [Burge, 1975], are another popular technique for functional programming. Parser combinators encourage small parsers to be represented as functions, and combined into larger ones with higher-order functions. Consequently, they ensure type safety in the host language mostly for parsing embedded domain-specific languages.

Normally, parser combinators are restricted to recursive descent parsing in a top-down manner. Although they bring great convenience for maintaining and manipulating parsers, there are two long-standing issues: lack of support for left-recursion, and bad efficiency with backtracking. And they have shown to be thorny even in the famous Parsec [Leijen and Meijer, 2001] library of Haskell. People have put great efforts in tackling those issues. As a representative, Packrat parsing [Ford, 2002] manages to support left-recursive grammars and guarantees linear-time complexity by memoization. It is a general algorithm that can be implemented in lazy functional languages.

Chapter 4 will focus closely on the Scala language. Scala has a standard Packrat-parsing library [1] of combinators for sequencing, choice, repetition and so on. Table 2.1 presents some common combinators with documentation. For better illustration, below is a simple example.

EXAMPLE We want to parse the expression language for literals and additions. The concrete syntax is shown below as a left-recursive grammar:

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

Table 2.1: Common combinators from the Scala standard parser combinator library.

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

```
<expr> ::= <int>
        |  <expr> '+' <expr>
```

Figure 2.12 defines the AST in a traditional OO style using traits and classes, and also defines the parser. The AST supports pretty-printing as a valid operation on expressions. In `ExpParser`, `Parser[E]` denotes the synonym for a Packrat parser, where `E` indicates the type of results it produces. `parse` is a generic function for testing. `lexical` represents the lexer. `pLit` parses an integer for the literal case. `pAdd` handles the addition case and creates an object of `Add`, where the two sub-expressions are parsed with a recursive call of `pExpr`. Finally, `pExpr` composes `pLit` and `pAdd`; the composition is achieved by `|||`, the longest-match alternative combinator which ignores the order of parser components. A test in `main` gives its output as expected:

```
println(ExpParser.parse("2 + 3 + 5").print) // (2 + (3 + 5))
```

2.6 Monad Transformers

Monads are widely considered as a practical and type-safe technique for encoding side effects in functional programming. They abstract computations at a high level with a couple of operations, thus simplifying client code significantly. Furthermore, *monad transformers*, firstly proposed by Liang et al. [1995] in an implementation of modular interpreters, are used to construct monads in a modular way. Transformers are rolled into a stack, composing those individual features together into full functionality, while each component is still easily accessible.

A monad transformer takes an underlying monad as a type parameter, and defines the modified behavior on such a monad. Haskell provides a monad transformer library (MTL), including `StateT` for mutable state, `ExceptT` for exceptions, `ContT` for continuations, and so on.

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

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

object ExpParser extends StandardTokenParsers with PackratParsers {
  type Parser[E] = PackratParser[E]
  def parse(input: String) = {
    val t = phrase(pExpr)(new lexical.Scanner(input))
    t.getOrElse(sys.error(t.toString))
  }

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

Figure 2.12: Parsing the expression language with Scala Packrat parsers.

EXAMPLE Our illustration takes such an operation as an example: we want to exhaustively traverse a directory, and list all included files in absolute paths. A conventional implementation requires the **IO** environment, together with `System.FilePath` and `System.Directory` packages. Figure 2.13 defines `listDir` for the directory walker. With the root directory `dir`, `listDirectory` collects all the sub-directories into a list. Then it checks whether each sub-directory is a single file, or a folder. The paths of files will be stored into the result, while folders are recursively traversed by `listDir`. Finally, **concat** merges all results into a single list.

An equivalent implementation uses monad transformers shown in Figure 2.14. As an input, the root directory can be encoded with a reader monad, while outputting file information refers to a writer monad. Consequently `listDir'` forms a single monadic value, with **IO** at the innermost position. In that definition, `ask` retrieves the input directory, and `local` runs a recursive `listDir'` with a new input, while `tell` simply dumps a file path to the writer. It can be proved that `listDir` behaves identically to

```
execWriter . runReaderT listDir'
```

Furthermore, monad transfomers contribute to code reusability. The above `listDir'` can instead have the following generic type signature

```
listDir :: FilePath -> IO [FilePath]
listDir dir = do
  dirs <- listDirectory dir
  liftM concat . forM dirs $ \d -> do
    let path = dir </> d
    isDir <- doesDirectoryExist path
    if isDir then listDir path else return [path]
```

Figure 2.13: Recursing a directory tree and output all files.

```
listDir' :: ReaderT FilePath (WriterT [FilePath] IO) ()
listDir' = do
  dir  <- ask
  dirs <- liftIO $ listDirectory dir
  forM_ dirs $ \d -> do
    let path = dir </> d
    isDir <- liftIO $ doesDirectoryExist path
    if isDir then local (\_ -> path) listDir' else tell [path]
```

Figure 2.14: Directory walker by monad transformers.

```
listDir' :: (MonadReader FilePath m,
             MonadWriter [FilePath] m,
             MonadIO m) => m ()
```

without changing the body. It implies adaptivity to future extensions of functionality.

2.7 QuickCheck

QuickCheck [Claessen and Hughes, 2000] is widely known as a combinator library for testing Haskell programs. Programmers write specifications for their programs, in the form of logical properties, and QuickCheck automatically generates a large number of test cases to check those properties. This section only introduces some features preliminary for understanding this dissertation.

PROPERTIES Properties are basically functions with return type `Bool`, but can also have `Property` type with conditionals and quantifiers provided by QuickCheck. For example:

```
prop_HeadLast :: Eq a => [a] -> Property
prop_HeadLast xs = not (null xs) ==> head (reverse xs)
```

The above proposition states that if a list is non-empty, taking the head element is equivalent to taking the last element of the reversed list. To check the property:

```
> quickCheck prop_HeadLast
+++ OK, passed 100 tests.
```

where the input lists are randomly generated by QuickCheck.

Generators for Custom Datatypes The precondition of letting QuickCheck randomly generate objects for a specific type, is to make the type an instance of `Arbitrary`. Besides the built-in instances for primitive types, users can also implement custom generators for their own datatypes. QuickCheck offers a number of auxiliary combinators, which deal with sized parameters, random distribution and so on.

For instance, an arithmetic language with literals and additions is represented by the following recursive datatype:

```
data Exp = Lit Int | Add Exp Exp
```

And the `Arbitrary` interface is defined as follows:

```
class Arbitrary a where
  arbitrary :: Gen a
  ...
```

where `Gen a` represents a generator for type-`a` values. For `Exp`, one implementation could be:

```
instance Arbitrary Exp where
  arbitrary = sized genExp
    where genExp n | n <= 0    = liftM Lit $ choose (0, 100)
                   | otherwise = do e1 <- genExp (n - 1)
                                    e2 <- genExp (n - 1)
                                    elements [Add e1 e2, Mul e1 e2]
```

Note that `sized` allows a generator to hold a size parameter. When the input `n` is non-positive, a literal is randomly picked from `0` to `100`. On the other hand, when `n` is greater than `0`, two sub-expressions are recursively generated from `genExp`, both with size `n - 1`. Then `elements` combines them by choosing between addition and multiplication with equal probability. Such a generator constructs a complete binary tree of a certain depth.

Chapter 3

Scrap Your Boilerplate with Object Algebras

This chapter presents *Shy*, a type-safe Java framework for the purpose of removing boilerplate code in traversing abstract syntax trees (ASTs). Complex ASTs are commonly used for processing real-life programming languages, with dozens or even hundreds of kinds of nodes. They typically require large amounts of traversal code, whereas in most situations, only a small portion implements the real functionality, leaving much boilerplate code that simply walks the structure.

Object Algebras [Oliveira and Cook, 2012a], briefly described in Section 2.2.3 as a solution to the Expression Problem, are the underlying mechanism of representing ASTs in *Shy*. As a result, *Shy* supports modular and type-safe extensibility on both ASTs and their traversal operations. In order to reduce boilerplate traversal code, *Shy* captures four different patterns of generic traversals: *queries*; *transformations*; *generalized queries*; and *contextual transformations*. These patterns are represented by Object Algebras, and are automatically generated from a simple Java annotation `@Algebra` put on top of ASTs. Programmers who want to implement structure-shy traversals can then inherit from one of these four Object Algebras, and override only the cases that deal with the interesting parts of the traversal. Consequently traversals written in *Shy* are:

- *Small in size:* With *Shy* the amount of code that programmers need to write a structure-shy traversal is significantly smaller. Often traversals in *Shy* are implementable in just a few lines of code, even for complex ASTs with hundreds of different types of nodes.
- *Adaptive and structure-shy:* Traversals written with *Shy* can omit boilerplate code, making traversals more adaptive to future changes or extensions to the data type.
- *Type-safe: Shy* traversals are directly written in Java and the Java type-system ensures type-safety. No run-time casts are needed for generic traversal code or for user-defined traversal code.
- *Extensible with separate compilation:* Traversals inherit type-safe extensibility from Object Algebras. Both traversals and the AST structures are extensible. Thus it is possible to reuse traversal code for ASTs extended with additional node types. Furthermore *Shy* traversals support separate compilation.

```
form DriverLicense {
  name: "What is your name?" string
  age: "What is your age?" integer
  if (age >= 18)
    license: "Have a driver's license?" boolean
}
```

Figure 3.1: Example QL questionnaire: driver's license.

- *Implemented in plain Java: Shy* traversals do not require a new tool or language. The approach is library based and only uses Java annotations.

To prove the effectiveness of the approach, *Shy* has been applied in the implementation of QL, a domain-specific language (DSL) for defining questionnaires that has been used before as a benchmark language [Erdweg et al., 2013; Gouseti et al., 2014]. The results show that for a large number of traversals there was a significant reduction in the amount of user-defined code: only 4% to 21% of the AST cases had to be implemented in comparison with code written without *Shy*.

This chapter is organized as follows: Section 3.1 gives the motivation and an overview of *Shy* with concrete examples. Section 3.2 to 3.5 illustrate in detail the four generic traversal patterns, in the context of Object Algebra encoding. Section 3.6 describes an additional scenario of traversals called *desugaring transformations*. Section 3.7 takes an insight into the extensibility from *Shy*. Section 3.8 briefly introduces the implementation of *Shy*. Finally the case study is presented in Section 3.9 to illustrate the utility of the *Shy* framework.

3.1 An Overview of *Shy*

This section starts by illustrating the problem of boilerplate code when implementing traversals of complex structures. It then shows how *Shy* addresses the problem using a combination of Object Algebras [Oliveira and Cook, 2012a] and Java annotations.

3.1.1 Traversing Object-Oriented ASTs

We start by introducing the problem of boilerplate code by considering a simplified variant of the QL language used in our case study [Gouseti et al., 2014], called MiniQL. Just like QL, MiniQL can be used to describe interactive questionnaires.

An example MiniQL program is shown in Figure 3.1. The questionnaire first asks for the user's name and age, and then, if the age is greater than or equal to 18, asks if the user has a driver's license. Because of the conditional construct, the last question will only appear when the user is actually eligible to have driver's license.

MiniQL's abstract syntax contains forms, statements (if-then and question) and expressions (only literals, variables and greater-than-or-equal). A traditional OO implementation adopts the *Interpreter* pattern, shown in Figure 3.2. A form (class `Form`) has a name and consists of a list of statements. Statements are conditionals (`If`) which contain an expression and a statement body, and questions

```
class Form {
   String name; List<Stmt> body;
   Set<String> usedVars() {
      Set<String> vars = new HashSet<>();
      body.forEach(s -> vars.addAll(s.usedVars()));
      return vars;
   }
}

class If extends Stmt {
   Exp cond; Stmt then;
   Set<String> usedVars() {
      Set<String> vars=new HashSet<>(cond.usedVars());
      vars.addAll(then.usedVars());
      return vars;
   }
}

class Question extends Stmt {
   String name, label, type;
   Set<String> usedVars() { return emptySet(); }
}

class Lit extends Exp {
   int n;
   Set<String> usedVars() { return emptySet(); }
}

class Var extends Exp {
   String x;
   Set<String> usedVars() { return singleton(x); }
}

class GEq extends Exp {
   Exp lhs, rhs;
   Set<String> usedVars() {
      Set<String> vars = new HashSet<>(lhs.usedVars());
      vars.addAll(rhs.usedVars());
      return vars;
   }
}
```

Figure 3.2: Implementing the "used variables" operation using traditional ASTs.

```
class UsedVars implements QLAlg<Set<String>, Set<String>, Set<String>> {
   Set<String> Form(String n, List<Set<String>> b) {
      Set<String> vars = new HashSet<>();
      b.forEach(s -> vars.addAll(s));
      return vars;
   }

   Set<String> If(Set<String> c, Set<String> t) {
      Set<String> vars = new HashSet<>(c);
      vars.addAll(t);
      return vars;
   }

   Set<String> Question(String n,String l,String t) {
      return Collections.emptySet();
   }

   Set<String> Lit(int x) {
      return Collections.emptySet();
   }

   Set<String> Var(String x) {
      return Collections.singleton(x);
   }

   Set<String> GEq(Set<String> l, Set<String> r) {
      Set<String> vars = new HashSet<>(l);
      vars.addAll(r);
      return vars;
   }
}
```

Figure 3.3: Implementing the "used variables" operation using Object Algebras.

(`Question`) which have a name, label and type. Expressions are standard, but limited to literals (`Lit`), variables (`Var`) and greater-than-or-equal (`GEq`).

BOILERPLATE IN QUERIES Queries are one typical pattern of traversals. Figure 3.2 already shows a query over MiniQL structures, namely the collection of used variables. The operation is defined using the method `usedVars`, declared in the abstract superclasses `Stmt` and `Exp` (omitted for brevity), and implemented in the concrete statement and expression classes. As can be seen, the only interesting bit of code is the `usedVars` method in class `Var`. All other implementations merely deal with aggregating results of their child nodes, or returning a default empty set.

```
@Algebra
public interface QLAlg<E, S, F> {
   F Form(String name, List<S> body);
   S If(E cond, S then);
   S Question(String name,String label,String type);
   E Lit(int n);
   E Var(String x);
   E GEq(E lhs, E rhs);
}
```

Figure 3.4: Object Algebra interface of the MiniQL abstract syntax.

BOILERPLATE IN TRANSFORMATIONS The boilerplate code exhibited in the `usedVars` query often also applies to transformations. Consider for example a rename transformation which takes a `Form` and returns another form where the occurrences of the specified variable are renamed. Again, the only interesting cases would be in the `Var` and `Question` classes, where the actual renaming is applied. All other classes, however, require boilerplate to recreate the structure. The full code in Appendix A.1 contains a simple example of such a rename operation as well.

In addition to the significant amount of boilerplate code, there is another drawback to the traditional OO solution, as discussed in Section 2.2.1, that it does not support extensibility along the dimension of operations. Each new operation requires pervasive changes across the AST classes.

3.1.2 Modeling MiniQL with Object Algebras

Figure 3.3 shows the used variables operation implemented using Object Algebras. The operation is a class implementing the Object Algebra interface (`QLAlg`) shown in Figure 3.4.

The `UsedVars` class provides an implementation for each of the methods in the Object Algebra interface, which together define the full used variables operation. Since the result of collecting those variables is `Set<String>`, all the type parameters are set to that type. Most of the method implementations simply traverse the child nodes and accumulate the variable names. That is the case, for example, for `Form`. Again, the only method implementation that does something different is `Var`, which returns the `x` argument.

Unlike the standard OOP implementation, Object Algebras support adding operations without changing existing code. For instance, the renaming operation mentioned above could be realized as follows:

```
class Rename<E, S, F> implements QLAlg<E, S, F> {
   QLAlg<E, S, F> alg;
   String from, to;

   F Form(String n, List<S> b) {
      return alg.Form(n, b);
   }
```

```
class UsedVars implements QLAlgQuery<Set<String>> {
   public Monoid<Set<String>> m() {
      return new SetMonoid<String>();
   }
   public Set<String> Var(String name) {
      return Collections.singleton(name);
   }
}
```

Figure 3.5: MiniQL used variables, implemented with *Shy*.

```
      S Question(String n, String l, String t) {
         n = n.equals(from) ? to : n;
         return alg.Question(n, l, t);
      }
      ...
      E Var(String x) {
         x = x.equals(from) ? to : x;
         return alg.Var(x);
      }
   }
```

Each constructor reconstructs a new node using an auxiliary MiniQL algebra `alg`. Almost all the method implementations reconstruct the structure with no changes using the methods of `alg`. For instance, the `Form` method just recreates the form in the algebra `alg`. The other boilerplate cases are omitted for brevity; the full code can be found in Appendix A.2. The two exceptions are the methods `Question` and `Var`, where the identifiers with the given name `from` are renamed to `to`.

Although the Object Algebra encoding of MiniQL solves the problem of extensibility, the traversal code still contains boilerplate code. In both `UsedVars` and `Rename`, the only interesting code is in a small number of cases. Ideally, we would like to write only the code for the interesting cases, and somehow "inherit" the tedious traversal code.

3.1.3 *Shy*: An Object Algebra Framework for Traversals

To deal with the boilerplate problem we created *Shy*: a Java Object Algebras framework, which provides a number of generic traversals at the cost of a single annotation. The key idea in *Shy* is to automatically create highly generic Object Algebras, which encapsulate common types of traversals. In particular *Shy* supports generic *queries* and *transformations*. The two types of traversals are, for instance, sufficient to capture the used variables and renaming operations.

Automatic Generation of Generic Traversals With *Shy*, programmers just need to add the `@Algebra` annotation to the definition of `QLAlg` to get the code for generic queries and transformations. An example of that annotation is already shown in Figure 3.4. Triggered by the annotation, *Shy* generates base traversal interfaces with Java 8 `default` methods which can then be overridden to implement specific behavior. For instance, for the MiniQL algebra, *Shy* generates interfaces `QLAlgQuery`

```
class Rename<E, S, F> extends QLAlgTrans<E, S, F> {
   String from, to;
   Rename(QLAlg<E,S,F> alg, String from, String to) {
      super(alg);
      this.from = from;
      this.to = to;
   }
   public S Question(String n, String l, String t) {
      n = n.equals(from) ? to : n;
      return qLAlg().Question(n, l, t);
   }
   public E Var(String x) {
      x = x.equals(from) ? to : x;
      return qLAlg().Var(x);
   }
}
```

Figure 3.6: MiniQL renaming, implemented with *Shy*.

and `QLAlgTrans` which can be used to implement `UsedVars` and `Rename` in only a fraction of the code[1].

The *Shy*-based implementation of both operations is shown in Figure 3.5 and 3.6. In contrast to Figure 3.3, the code in Figure 3.5 is much shorter. By implementing the `QLAlgQuery` and `QLAlgTrans` interface, only the methods `Question` and `Var` need to be overridden: all the other methods perform basic accumulation for queries and basic reconstruction in the case of transformations. For queries the only extra thing a programmer has to do is to provide an instance of a monoid, which is used to specify how to accumulate the results during the traversal. Similarly, for transformations, the programmer needs to pass an algebra for providing the constructors for creating the result of a transformation.

Client Code To use the queries and operations on a questionnaire like the one in Figure 3.1, we need a function to create a structure using the generic MiniQL interface:

```
<E, S, F> F makeQL(QLAlg<E, S, F> alg) {
   return alg.Form("DriverLicense", Arrays.asList(
      alg.Question("name", "Name?", "string"),
      alg.Question("age", "Age?", "integer"),
      alg.If(alg.GEq(alg.Var("age"), alg.Lit(18)),
             alg.Question("license", "License?", "boolean"))));
}
```

Since both queries and transformations are implementations of the MiniQL interface, they can be passed to the `makeQL` function defined above:

```
println(makeQL(new UsedVars()));
```

[1]The generated code is available in Appendix A.3 and A.4.

```
println(makeQL(new Rename<>(new UsedVars(), "age", "AGE")));
```

This code prints out `[age]` and `[AGE]`, which are the set of used variables before and after renaming, respectively. Note how the `Rename` transformation transforms the questionnaire into the `UsedVars` algebra.

The remainder of this chapter provides the details and implementation techniques used in *Shy*. Besides basic queries and transformations, *Shy* also supports two generalizations of these types of traversals called *generalized queries* and *contextual transformations*.

3.2 Queries

This section shows the ideas behind generic queries and how they are implemented in *Shy*. A query is an operation that traverses a structure and computes some aggregate value. The inspiration for queries comes from similar types of traversals used in functional programming libraries, such as "Scrap your Boilerplate" [Lammel and Jones, 2003].

AN EXPRESSION LANGUAGE The following code shows a simple expression language represented as the Object Algebra interface `ExpAlg`.

```
@Algebra
interface ExpAlg<Exp> {
   Exp Var(String s);
   Exp Lit(int i);
   Exp Add(Exp e1, Exp e2);
}
```

We will use this minimal Object Algebra interface throughout the rest of the chapter to illustrate the various different types of traversals supported by *Shy*. Three different kinds of nodes exist: a numeric literal, a variable or the addition of two expressions. Queries are illustrated by implementing an operation to compute the free variables in an expression.

3.2.1 Boilerplate Queries

Figure 3.7 shows a standard approach for computing free variables using Object Algebras[1]. A set of strings is used to collect the names of the free variables. The `Var` method returns a singleton set of `s`, whereas the `Lit` method returns an empty set. The more interesting case is in the `Add` method, where the two sets are joined into one.

The typical pattern of a query is to collect some information from some of the nodes of the structure, and to aggregate the information that comes from multiple child nodes. For example, in the case of free variables, the strings from the `Var` nodes are collected, and in the `Add` nodes the information from multiple children is merged into a single set.

An important observation about queries is that the code to aggregate information tends to be the same: if we had a subtraction node, the code would be essentially identical to `Add`. Moreover, there are

[1]Here and in the following we will use interfaces with *default methods* (as introduced in Java 8) to combine queries and transformations using multiple inheritance.

```
interface FreeVars extends ExpAlg<Set<String>> {
   default Set<String> Var(String s) {
      return Collections.singleton(s);
   }
   default Set<String> Lit(int i) {
      return Collections.emptySet();
   }
   default Set<String> Add(Set<String> e1, Set<String> e2) {
      return Stream.concat(e1.stream(),e2.stream()).collect(Collectors.toSet());
   }
}
```

Figure 3.7: Free variables as an Object Algebra.

only very few types of nodes that contain relevant information for the query. For nodes that contain information that is not relevant to the query, we simply return a neutral value (such as the empty set in `Lit`). Nonetheless, a programmer has to write this boring boilerplate code handling the traversals. While for the small structure presented here this may not look too daunting, in a large structure with dozens or even hundreds of constructors such code becomes a significant burden.

3.2.2 Generic Queries

A better approach would be to abstract the generic traversal and accumulation code for queries. This way, when programmers need to implement query operations, they can simply reuse the generic traversal code and focus only on dealing with the nodes that do something interesting.

The code that captures the aggregation and collection of information can be captured by a well-known algebraic structure called a *monoid*. Monoids are commonly used in functional programming for such purposes, but they are perhaps less commonly known in object-oriented programming. The interface of a monoid is defined as follows:

```
interface Monoid<R> {
   R join(R x, R y);
   R empty();
}
```

Intuitively, the `join()` method is used to combine the information from substructures, and `empty()` is an indicator of "no information".

GENERIC QUERIES Using the monoid operations alone, it is possible to write a generic query. Figure 3.8 shows how this is achieved. In nodes that contain child nodes, such as `Add`, the information is aggregated using `join`. In nodes that contain other information, such as `Var` and `Lit`, the query returns `empty`. This allows concrete queries to be implemented by overriding methods from multiple, different algebras.

```
interface ExpAlgQuery<Exp> extends ExpAlg<Exp> {
   Monoid<Exp> m();
   default Exp Var(String s)       { return m().empty(); }
   default Exp Lit(int i)          { return m().empty(); }
   default Exp Add(Exp e1, Exp e2) { return m().join(e1, e2); }
}
```

Figure 3.8: Generic queries using a monoid.

3.2.3 Free Variables with Generic Queries

The `ExpAlgQuery` interface provides an alternative way to define the free variables operation. Instead of directly defining the free variables operation, `ExpAlgQuery` can be inherited, provided that the method `m()` is implemented. In the case of free variables, the monoid returned by `m()` is an implementation of the `Monoid` interface for sets:

```
class SetMonoid<X> implements Monoid<Set<X>> {
   public Set<X> empty() {
      return Collections.emptySet();
   }
   public Set<X> join(Set<X> x, Set<X> y) {
      Set<X> tmp = new HashSet<>(x);
      tmp.addAll(y);
      return tmp;
   }
}
```

The method `empty()` corresponds to the empty set, and `join()` is implemented as union. Using this monoid the free variables operation is defined as follows:

```
interface FreeVars extends ExpAlgQuery<Set<String>> {
   default Monoid<Set<String>> m() {
      return new SetMonoid<String>();
   }
   default Set<String> Var(String s) {
      return Collections.singleton(s);
   }
}
```

There are two important differences to the implementation in Figure 3.7. The first difference is that the monoid to be used needs to be specified. However, the code for the monoid is still quite short (only requires two method implementations) and, more importantly, it is highly reusable. Indeed the `SetMonoid` is reused on various examples of queries. Because monoid instances are so general purpose, the *Shy* library already contains many common monoid implementations. Users do not usually have to define these instances themselves. The second difference is that now only the case for variables needs to be defined: the other cases are inherited from `ExpAlgQuery`.

Generic Template for Queries The traversal code in `ExpAlgQuery` is entirely mechanical and can be automatically generated. This is precisely what *Shy* does. Annotating algebra interfaces, such as `ExpAlg`, with the annotation `@Algebra`, triggers automatic generation of generic query interfaces, such as `ExpAlgQuery`. The general template in *Shy* for an algebra `Alg<`X_1`, ..., `X_n`>`, with constructors `f`$_1$`, ..., f`$_m$ is shown next:

```
interface Alg_Q<R> extends Alg<R,...,R> {
   Monoid<R> m();

   default R f_i() {
      return m().empty();
   }

   default R f_j(R p_1, ..., R p_k) {
      return  m().join(p_1, m().join(p_2, ..., m().join(p_{k-1}, p_k)...)));
   }
   ...
}
```

Note that interface `Alg`$_Q$ extends `Alg` so that all type parameters are unified as type `R`. All arguments to a constructor `f`$_j$ are combined with `join` from the monoid `m()`. Arguments with primitive types, like `int`, `boolean` or `String`, are ignored by default.

3.3 Generalized Queries

The previous section introduced simple queries where each constructor contributes to a single monoid. Recursive data types, however, often have multiple syntactic categories, for instance expressions and statements. In such multi-sorted Object Algebras each sort is represented by a different type parameter in the algebra interface. In this section we present *generalized queries*, where each such type parameter can be instantiated to different monoids. It turns out that, for some operations, this generalized version of queries is needed.

Example: Data Dependencies A simple example of a generalized query is the extraction of the data dependencies between assignment statements and variables in simple imperative programs. To express this query, the simple `ExpAlg` is first extended with statements using the `StatAlg` interface defined as follows:

```
@Algebra
public interface StatAlg<Exp, Stat> {
   Stat Seq(Stat s1, Stat s2);
   Stat Assign(String x, Exp e);
}
```

The `StatAlg` interface defines statement constructors for sequential composition (`Seq`) and assignment (`Assign`). The interface does not extend the `ExpAlg` interface; we rely on multiple inheritance of Java interfaces to combine implementations of these interfaces later (see Figure 3.10).

```
interface G_StatAlgQuery<Exp, Stat> extends StatAlg<Exp, Stat> {
   Monoid<Exp> mExp();
   Monoid<Stat> mStat();

   default Stat Assign(String x, Exp e) {
      return mStat().empty();
   }
   default Stat Seq(Stat s1, Stat s2) {
      return mStat().join(s1, s2);
   }
}
```

Figure 3.9: Default implementation of generalized queries over many-sorted statement algebra.

GENERALIZED QUERIES The generated default implementation of queries over statements is shown in Figure 3.9, while the generated code for expressions (`G_ExpAlgQuery`) is presented in Appendix A.5. Note that the interface declares two monoids, one for each sort. Since the `Assign` and `Seq` constructors create statements, they return elements of the `mStat()` monoid. Furthermore, because it is impossible to automatically join a monoid over one type with a monoid over another type, the `e` argument in `Assign` is ignored. As a result, a concrete implementation normally has to override this case to deal with the transition from expressions to statements.

Data dependencies are created by assignment statements: for a statement `Assign(String x, Exp e)` method, the variable `x` will depend on all variables appearing in `e`. The result of extracting such dependencies can be represented as binary relation (a set of pairs). In expressions we need to collect the free variables, which can be stored in a set of strings. Thus in this traversal two monoids are involved: a monoid for a set of pairs of strings; and a monoid for a set of strings.

To implement the extraction of data dependencies only two cases have to be implemented: the variable (`Var`) case from the `ExpAlg` signature; and the assignment (`Assign`) case from the `StatAlg` signature. The implementation is shown in Figure 3.10. Note that the `Assign` case takes the input `Set <String> e` and uses it to create the dependency relation. The propagation of dependencies across sequential composition is automatic, as is the propagation of the set of variables through the different types of expressions.

CLIENT CODE A structure using the generic interfaces `ExpAlg` and `StatAlg` is created as follows:

```
<E, S, A extends ExpAlg<E> & StatAlg<E, S>> S makeStat(A a) {
   return a.Seq(a.Assign("x", a.Add(a.Var("x"), a.Lit(3))),
                a.Assign("y", a.Add(a.Var("x"), a.Var("z"))));
}
```

Note that here the argument of `makeStat` must implement both `ExpAlg` and `StatAlg`. To achieve this in Java, `makeStat` has a type parameter `A` which is required to implement both interfaces. Using `makeStat` we can pass an instance of `DepGraph` to compute the dependencies of the statement.

```
println(makeStat(new DepGraph(){}));
```

```
interface DepGraph extends
      G_ExpAlgQuery<Set<String>>,
      G_StatAlgQuery<Set<String>, Set<Pair<String, String>>> {
   default Monoid<Set<String>> mExp(){
      return new SetMonoid<>();
   }
   default Monoid<Set<Pair<String, String>>> mStat(){
      return new SetMonoid<>();
   }
   default Set<String> Var(String x){
      return Collections.singleton(x);
   }
   default Set<Pair<String, String>> Assign(String x, Set<String> e){
      Set<Pair<String, String>> deps = new HashSet<>();
      e.forEach(y -> deps.add(new Pair<>(x, y)));
      return deps;
   }
}
```

Figure 3.10: Dependency graph with a generalized query.

The result is `[<x, x>, <y, x>, <y, z>]`, as expected.

3.4 Transformations

Queries are a way to extract information from a data structure. Transformations, on the other hand, allow data structures to be transformed into new structures. Just as with queries, we can distinguish code that deals with traversing the data structure from code that actually transforms the structure. In this section we show how to avoid most traversal boilerplate code in the context of transformations using *Shy*.

3.4.1 Transformations, Object Algebra Style

A simple example of a transformation algebra, using the Object Algebra interface `ExpAlg`, is substituting expressions for variables. A manual implementation based on Object Algebras is shown in Figure 3.11.

The expression to be substituted, and the variable to substitute for are provided by the methods `e()` and `x()` respectively. The method `expAlg()` is an instance of `ExpAlg` on which the transformation is based. Since Object Algebras are factories, the transformation is executed immediately during construction of tree structures. For instance, calling `Var("x")` on a `SubstVar` object with `x()` returning `"x"` immediately returns the result of `e()` (the original variable expression is never created). In the other cases, the original structure is recreated in the algebra `expAlg()`.

The following code shows how the transformation could be used:

```
interface SubstVar<Exp> extends ExpAlg<Exp> {
   ExpAlg<Exp> expAlg();
   String x(); Exp e();
   default Exp Var(String s) {
      return s.equals(x())? e(): expAlg().Var(s);
   }
   default Exp Lit(int i) { return expAlg().Lit(i); }
   default Exp Add(Exp e1, Exp e2) {
      return expAlg().Add(e1, e2);
   }
}
```

Figure 3.11: A normal algebra-based implementation of variable substitution.

```
FreeVars fv = new FreeVars() {};
SubstVar<Set<String>> subst = new SubstVar<Set<String>>() {
   public ExpAlg<Set<String>> expAlg() {
      return fv;
   }
   public String x() { return "x"; }
   public Set<String> e() {
      return fv.Add(fv.Lit(1),fv.Var("y"));
   }
};
Set<String> res = subst.Var("x");
```

The `SubstVar` interface is instantiated with `expAlg()` returning an instance of the `FreeVars` algebra defined earlier (e.g., Figure 3.7). The `x()` method returns the variable to be substituted (`"x"`). Finally, the `e()` returns a new expression `1 + y` over the `fv` algebra. When expressions are created on the `subst` algebra, the result is the set of free variables *after* the substitution has taken place. As a result, `res` will contain only `"y"`.

Note that this allows pipelining of transformations: there is no reason `expAlg()` cannot return yet another transformation algebra, for instance, a another instance of `SubstVar` realizing a different substitution. We elaborate on composing transformations this way in Section 3.9.2.

Unfortunately, we again observe the problem of traversal-only boilerplate code: the `Lit` and `Add` methods of Figure 3.11 simply delegate to the base algebra `expAlg()`, without doing any real work.

3.4.2 Generic Traversal Code

The boilerplate code in transformations can be avoided by creating a super-interface containing default methods performing the traversal (shown in Figure 3.12). A concrete transformation can then selectively override the cases of interest. Variable substitution can now be implemented as follows:

```
interface SubstVar<Exp> extends ExpAlgTransform<Exp> {
   String x(); Exp e();
```

```
interface ExpAlgTransform<Exp> extends ExpAlg<Exp> {
   ExpAlg<Exp> expAlg();
   default Exp Var(String s) {
      return expAlg().Var(s);
   }
   default Exp Lit(int i) { return expAlg().Lit(i); }
   default Exp Add(Exp e1, Exp e2) {
      return expAlg().Add(e1, e2);
   }
}
```

Figure 3.12: Traversal-only base interface for implementing transformations of expressions.

```
interface Alg_T<X_1,...,X_n> extends Alg<X_1,...,X_n> {
   Alg<X_1, ..., X_n> alg();

   default X_i f_j(X_p^1 p_1, ..., X_p^k p_k) {
      return alg().f_j(p_1, ..., p_k);
   }
   ...
}
```

Figure 3.13: Generic template for generating boilerplate of transformations.

```
      default Exp Var(String s) {
         return s.equals(x())? e(): expAlg().Var(s);
      }
   }
```

In this case, only the method `Var()` is overridden.

Generic Template for Transformations Just like in the case of queries, the traversal code in `ExpAlgTransform` is entirely mechanical and can be automatically generated by *Shy*. Figure 3.13 shows the general template for the generated code. Here $\texttt{Alg}_T$ extends `Alg` with the same type parameters and the base algebra `alg()` is declared inside.

3.5 Contextual Transformations

The previous section introduced a simple template for defining transformations. Transformations in this style may only depend on global context information (e.g., `x()`, `e()`). Many transformations, however, require context information that might change during the traversal itself. In this section we instantiate algebras over function types to obtain transformations which pass information down during traversal. Instead of having the algebra methods delegate directly to base algebra (e.g., `expAlg()`), this now happens indirectly through closures that propagate the context information.

```
interface Alg_CT<C,X_1,...,X_n> extends Alg<Function<C,X_1>,...,Function<C,X_n>> {
   Alg<X_1, ..., X_n> alg();

   default Function<C, X_i> f_j(Function<C,X_p^1> p_1, ..., Function<C,X_p^k> p_k) {
      return (c) -> alg().f_j(p_1.apply(c), ..., p_k.apply(c));
   }
   ...
}
```

Figure 3.14: Generic template for generating boilerplate of contextual transformations.

GENERIC TEMPLATE FOR CONTEXTUAL TRANSFORMATIONS Figure 3.14 shows the general template for an `Alg<X1, ..., Xn>`, with constructors `f1, ..., fm`. Note that interface $\texttt{Alg}_{CT}$ extends `Alg` and instantiates the type parameters to `Functions` from the context argument `C` to the corresponding sort $\texttt{X}_i$. Each constructor method now creates an anonymous function which, when invoked, calls the functions received as parameters ($\texttt{p}_1$ to $\texttt{p}_k$) and only then creates a structure over the `alg()` algebra.

EXAMPLE: CONVERSION TO DE BRUIJN INDICES An example of a contextual transformation is converting variables to De Bruijn indices in the lambda calculus [De Bruijn, 1972]. Using De Bruijn indices, a variable occurrence is identified by a natural number equal to the number of lambda terms between the variable occurrence and its binding lambda term. Lambda terms expressed using De Bruijn indices are useful because they are invariant with respect to alpha conversion.

The conversion to De Bruijn indices uses an object algebra interface `LamAlg` with constructors for lambda abstraction (`Lam`) and application (`Apply`). See below:

```
@Algebra
public interface LamAlg<Exp> {
   Exp Lam(String x, Exp e);
   Exp Apply(Exp e1, Exp e2);
}
```

Shy automatically generates the traversal code for transformation for both `LamAlg` and `ExpAlg`: `G_LamAlgTransform` and `G_ExpAlgTransform`, respectively. The generated transformation code can be found in Figure 3.15, where the auxiliary function `cons` for inserting an element to the head of a list is defined as follows:

```
public class Util {
   public static <X> List<X> cons(X x, List<X> l) {
      l = new ArrayList<>(l);
      l.add(0, x);
      return l;
   }
}
```

```
public interface G_ExpAlgTransform<A, B0> extends ExpAlg<Function<A, B0>> {
   ExpAlg<B0> expAlg();
   default <B> List<B> substListExpAlg(List<Function<A, B>> list, A acc) {
      List<B> res = new ArrayList<B>();
      for (Function<A, B> i : list)
         res.add(i.apply(acc));
      return res;
   }
   default Function<A, B0> Add(Function<A, B0> p0, Function<A, B0> p1) {
      return acc -> expAlg().Add(p0.apply(acc), p1.apply(acc));
   }
   default Function<A, B0> Lit(int p0) {
      return acc -> expAlg().Lit(p0);
   }
   default Function<A, B0> Var(java.lang.String p0) {
      return acc -> expAlg().Var(p0);
   }
}

public interface G_LamAlgTransform<A, B0> extends LamAlg<Function<A, B0>> {
   LamAlg<B0> lamAlg();
   default <B> List<B> substListLamAlg(List<Function<A, B>> list, A acc) {
      List<B> res = new ArrayList<B>();
      for (Function<A, B> i : list)
         res.add(i.apply(acc));
      return res;
   }
   default Function<A, B0> Apply(Function<A, B0> p0, Function<A, B0> p1) {
      return acc -> lamAlg().Apply(p0.apply(acc), p1.apply(acc));
   }
   default Function<A, B0> Lam(java.lang.String p0, Function<A, B0> p1) {
      return acc -> lamAlg().Lam(p0, p1.apply(acc));
   }
}
```

Figure 3.15: Generated default contextual transformations of `ExpAlg` and `LamAlg`.

Using these interfaces, the conversion to De Bruijn indices is realized as shown in Figure 3.16. Note again that only the relevant cases are overridden: `Var` (from `ExpAlg`) and `Lam` (from `LamAlg`).

Client Code A structure using the generic interfaces `ExpAlg` and `LamAlg` is created as follows:

```
<E, A extends ExpAlg<E> & LamAlg<E>> E makeLamExp(A alg) {
   return alg.Lam("x", alg.Lam("y", alg.Add(alg.Var("x"), alg.Var("y"))));
}
```

```
interface DeBruijn<E> extends G_ExpAlgTransform<List<String>, E>,
                              G_LamAlgTransform<List<String>, E> {
   default Function<List<String>, E> Var(String p0) {
      return xs -> expAlg().Var("" + (xs.indexOf(p0) + 1));
   }

   default Function<List<String>,E> Lam(String x, Function<List<String>, E> e) {
      return xs -> lamAlg().Lam("", e.apply(cons(x, xs)));
   }
}
```

Figure 3.16: Converting variables to De Bruijn indices.

```
class PrintExpLam implements ExpAlg<String>, LamAlg<String> {
   public String Lam(String x, String e) {
      return "\\" + x + "." + e;
   }
   public String Apply(String e1, String e2) {
      return "(" + e1 + " " + e2 + ")";
   }
   public String Var(String s) { return s; }
   public String Lit(int i) { return i + ""; }
   public String Add(String e1, String e2) {
      return "(" + e1 + " + " + e2 + ")";
   }
}
```

Figure 3.17: Pretty-printing lambda expressions.

It simply generates "$\lambda x.\lambda y.\ x + y$" as a generic lambda expression. By instantiating the `DeBruijn` interface with a `PrintExpLam` algebra (shown in Figure 3.17) passed in as the base algebra, we can write the client code as follows:

```
DeBruijn<String> deBruijn = new DeBruijn<String>() {
   PrintExpLam alg = new PrintExpLam();
   public ExpAlg<String> expAlg() { return alg; }
   public LamAlg<String> lamAlg() { return alg; }
};
println(makeLamExp(deBruijn).apply(Collections.emptyList()));
```

The printed output is "`\.\.(2 + 1)`", which corresponds to original lambda term, but without variables.

3.6 Desugaring Transformations

In Section 3.4, we presented transformations, as well as the generic traversals generated by *Shy*. Although different constructors can be used in transforming a data structure, the generic transformations generated by *Shy* are type-preserving: they transform structures over one type (e.g., expressions) to different structures in the same type.

Desugaring transformations eliminate syntactic constructs by transforming them to a combination of constructs in a smaller base language. In this section we describe how *Shy* can be applied to implement compositional desugarings in a type-safe and extensible manner. In particular, the type system will enforce that the resulting language is indeed "smaller", and that consequently the desugared construct is guaranteed to be fully eliminated.

As an example, consider extending `ExpAlg` with a doubling construct which multiplies its argument expression by two. The Object Algebra interface that implements doubling is defined as follows:

```
@Algebra
public interface DoubleAlg<E> {
   E Double(E e);
}
```

An expression `Double(e)` can be desugared to `Add(e,e)`. The following code realizes this transformation by extending the `ExpAlgTransform` interface, generated by *Shy*:

```
interface Desugar<E> extends DoubleAlg<E>, ExpAlgTransform<E>  {
   default E Double(E e) {
      return expAlg().Add(e, e);
   }
}
```

The interface `Desugar` exports the language `DoubleAlg` and `ExpAlg`, but `expAlg()` (which is used as a factory for output expressions) has type `ExpAlg`. Since `ExpAlg` does not contain `Double`, the `Double` constructor cannot be used to construct the output. As a result, the algebra `Desugar` transforms into is guaranteed to be without any occurrences of `Double`.

Client Code Expressions are created over the combined languages `DoubleAlg` and `ExpAlg`:

```
<E, Alg extends DoubleAlg<E> & ExpAlg<E>> E makeExp(Alg a) {
   return a.Add(a.Lit(5), a.Double(a.Var("a")));
}
```

To illustrate the use of the `Desugar` algebra, here is the code to print an expression in desugared form:

```
ExpAlg<String> print = new PrintExp();
Desugar<String> desugar = new Desugar<String>() {
   @Override
   public ExpAlg<String> expAlg() {
      return print;
   }
};
```

```
System.out.println(makeExp(desugar));
```

The kind of desugarings presented in this section are limited to bottom-up, compositional desugaring, corresponding to factory methods directly invoking methods in a different algebra. As a result, these desugarings are executed in a bottom-up fashion: the arguments are always desugared before an expression itself is transformed. Because the transformations are generic with respect to the carrier object of the argument (as indicated by the type parameter `E`) it is impossible to look at the arguments. This prevents desugarings to perform complex, deep pattern matching on the argument structure. An added benefit, however, is that the desugaring is automatically deforested: intermediate expression trees are never created.

3.7 Extensible Queries and Transformations

Shy queries and transformations inherit modular extensibility from the Object Algebra design pattern. New transformations or queries are simply added by extending the interfaces generated by *Shy*. More interestingly, however, it is also possible to extend the data type with new constructors. Here we briefly describe how queries and transformations can be extended in this case.

3.7.1 Linear Extensibility

Consider again the extension of the expression language with lambda and application constructs (cf. Section 3.5). This requires changing the free variables query, since variables bound by `Lam` expressions need to be subtracted from the set of free variables of the body. Instead of reimplementing the query from scratch, it is possible to modularly extend the existing `FreeVars` query:

```
interface FreeVarsWithLambda extends FreeVars, LamAlgQuery<Set<String>> {
   default Set<String> Lam(String x,Set<String> f) {
      return f.stream().filter(y -> !y.equals(x)).collect(toSet());
   }
}
```

The interface `FreeVarsWithLambda` extends both the original `FreeVars` query and the base query implementation that was generated for the `LamAlg` interface defining the language extension. Note again, that only the relevant method (`Lam`) needs to be overridden.

For transformations the pattern is similar. To illustrate extension of transformation, consider the simple transformation that makes all variable occurrences unique, to distinguish multiple occurrences of the same name:

```
interface Unique<E> extends ExpAlgTransform<E> {
   int nextInt();
   default E Var(String s) {
      return expAlg().Var(s + nextInt());
   }
}
```

The `Unique` transformation uses a helper method `nextInt` which returns consecutive integers on each call. The basic transformation simply renames `Var` expressions. If, again, the expression

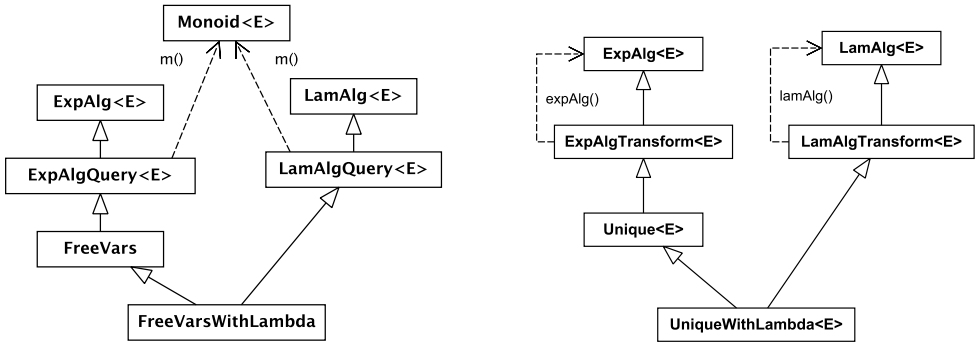

Figure 3.18: Extension of the `FreeVars` query (left) and the `Unique` transformation (right).

language is extended with lambda constructs, the transformation needs to be updated as well to make the variable in the binding position of lambda expression unique. The following code shows how this can be done in a modular fashion:

```
interface UniqueWithLambda<E> extends Unique<E>, LamAlgTransform<E> {
   default E Lam(String x, E e) {
      return lamAlg().Lam(x + nextInt(), e);
   }
}
```

Note that the transformation uses the `lamAlg()` algebra (from `LamAlgTransform`), to create lambda expressions.

Figure 3.18 gives a high level overview of query and transformation extension using the examples for `FreeVars` and `Unique`, respectively. In the case of queries, the abstract `m()` method will be shared by both the `FreeVars` and `FreeVarsWithLambda` interfaces. On the other hand, transformations are based on multiple base algebras, for sets of data type constructors (e.g., `expAlg()` and `lamAlg()`).

Note finally that, in the current implementation of *Shy* transformations, it is assumed that the language signatures `ExpAlg` and `LamAlg` are completely independent. This is however, not an essential requirement. An alternative design could have `LamAlg` be a proper extension of `ExpAlg` (i.e. `LamAlg <E> extends ExpAlg<E>`). In that case, the generated `LamAlgTransform` would need to refine the return type of the `expAlg()` method.

3.7.2 Independent Extensibility

Figure 3.18 shows extensions of queries and transformations where the extensions directly inherit from the concrete implementation of the operations (i.e. `FreeVars` and `Unique`, respectively). It is also possible to make the extensions more independent using multiple inheritance of interfaces with default methods.

For instance, the `Unique` transformation for the `LamAlg` language could also have been implemented independently of `Unique`:

```
interface UniqueLam<E> extends LamAlgTransform<E> {
```

```
    int nextInt();
    default E Lam(String x, E e) {
       return lamAlg().Lam(x + nextInt(), e);
    }
}
```

Note that this interface declares the required dependency on `nextInt()`. Both transformations are combined independently through multiple inheritance of interfaces. Both `Unique` and `UniqueLam` declare the method `nextInt()`, but since the declarations are abstract, they are identified. Implementations of the combined interface need to provide concrete implementations for `nextInt()`, `expAlg()` and `lamAlg()`:

```
class Combine<E> implements Unique<E>, UniqueLam<E> {
  public int nextInt() { ... }
  public ExpAlg<E> expAlg() { ... }
  public LamAlg<E> lamAlg() { ... }
}
```

For queries the pattern is the same, except that only a concrete implementation of `m()` has to be provided for the combined interface.

3.8 *Shy* Implementation

Shy is implemented using the standard Java annotations framework (`javax.annotation`) packaged in the Java Development Kit. All of the generic traversals are automatically generated by *Shy* for Object Algebra interfaces annotated with `@Algebra`. When an Object Algebra interface is annotated with `@Algebra`, *Shy* retrieves the required information from the interface (such as the names and types of factory methods) using reflection. *Shy* then generates the code based on the templates shown earlier. Furthermore, the *Shy* framework includes the `Monoid` interface as well as several useful implementations of it.

A major advantage of using standard Java annotations is that the code generation of the generic traversals can be done transparently: users do not need to use or install a tool to generate that code. As a result *Shy* is as simple to use as a conventional library. With little configuration effort, the code generation is automatically enabled in IDEs like Eclipse or IntelliJ. Finally, the framework is very small (around 885 source lines of code), so it can be easily be customized, if needed.

3.9 Case Study

To illustrate the utility of *Shy* we have implemented a number of queries and transformations in the context of QL, a DSL for questionnaires which has been implemented using Object Algebras before [Gouseti et al., 2014]. QL is similar to MiniQL, except that it additionally features an if-then-else construct, computed questions (which will appear read only), and a richer expression language. For more information on the features of QL we refer to [Erdweg et al., 2013].

Table 3.1: Number of overridden cases per query and transformation in the context of the QL implementation.

Operation	Template	Exp (18)	Stmt (5)	Form (1)	%
Collect variables	query	1			4%
Type environment	query		2		8%
Data dependencies	generalized query		3	1	17%
Control dependencies	generalized query		4	1	21%
Rename variable	transformation	1	2		13%
Inline conditions	contextual transformation		4		17%
Desugar "repeat"	contextual transformation	1	3		16%
Desugar "unless"	desugaring transformation		1		4%

3.9.1 QL Queries and Transformations

The queries extract derived information from a QL program, such as the set of used variables, the data and control dependencies between questions, and the global type environment. The transformations include two transformations of language extensions to the base language. The first realizes a simple desugaring of "unless(c)..." to "if(not(c))...". The second desugaring statically unfolds a constant bound loop construct ("repeat (i)...") and renames variables occurring below it accordingly. Finally, we have implemented a simple rename variable operation, and a flattening normalizer which inlines the conditions of nested if-then constructs.

Table 3.1 shows the number of cases that had to be overridden to implement each particular operation. The top row shows the number of constructs for each syntactic category in QL (Exp, Stmt, and Form). As can be seen, none of the operations required implementing all cases. The last column shows the number of overridden cases as a percentage. For this set of queries and transformations, almost no expression cases needed to be overridden, except the "Var" case in collect variables, rename variable and desugar "repeat"[1]. The cases required for desugaring include the case of the language extension (e.g. `Unless` and `Repeat`, respectively). These cases are not counted in the total in the first row but are used to compute the percentage.

3.9.2 Chaining Transformations

A typical compiler consists of many transformations chained together in a pipeline. *Shy* transformations support this pattern by passing transformation algebras as the base algebra to the implementation of another transformation. For instance, the desugar unless transformation desugars the "unless" statement to "if" statements in another algebra. The latter can represent yet another transformation.

In the context of QL, "unless" desugaring, condition inlining and variable renaming can be chained together as follows:

```
alg = new Desugar<>(
        new Inline<>(
          new Rename<>(Collections.singletonMap("x",  "y"), new Format())));
```

[1]Note, however, that the dependency extraction queries reuse the collect variables query on expressions.

The chained transformation `alg` first desugars "unless", then inlines conditions, and finally renames xs to ys. The `Rename` transformation gets as base algebra an instance of `Format`, a pretty printer for QL.

The algebra `alg` can now be used to create questionnaires:

```
Function<IFormatWithPrecedence, IFormat> pp = alg.form("myForm",
    Arrays.asList(alg.unless(alg.var("x"),
                  alg.question("x", "X?", new TBoolean()))));
```

Since inlining is a contextual transformation, the result of constructing this simple questionnaire is a function object representing the "to be inlined" representation of the questionnaire after desugaring. The `IFormatWithPrecedence` and `IFormat` types are formatting operations, respectively representing expressions and statements; these types originate from the `Format` algebra passed to `Rename`.

Calling the function with a boolean expression representing **`true`** will trigger inlining of conditions and renaming. The result is then a formatting object (`IFormat`) which can be used to print out the transformed questionnaire:

```
form myForm { if (true && !y) y "X?" boolean }
```

As can be seen, the variable `x` has been renamed to `y`. The (renamed) condition `y` is now negated, because of the desugaring of "unless". Finally, the result of inlining conditions can be observed from the conjunction in the **`if`** statement.

3.9.3 *Shy* Performance vs Vanilla ASTs

We compared the performance characteristics of the operations implemented using *Shy* with respect to vanilla implementations based on ordinary AST classes with ordinary methods representing the transformations and queries. In the vanilla implementation, the program was parsed into an AST structure, and then the operation was invoked and measured. In the case of the *Shy* queries, constructing the "AST" corresponds to executing the query, so we measured that. For context-dependent transformations, however, building the "AST" corresponds to constructing the function to execute the transformation, hence we only measured invoking this function. The vanilla query implementations use the same monoid structures as in *Shy*.

The operations were executed on progressively larger QL programs (up to 140Kb). The QL programs represent questionnaires describing a binary search problem (a number guessing game) and are automatically generated, with increasing search spaces. The benchmarks were executed on a 2.6GHz MacBook Pro Intel Core i5 with 8GB memory. The JVM was run in 64bit server mode and was given 4Gb of heapspace to minimize the effect of garbage collection pauses. Each benchmark was run without measuring first, to warm-up the JVM. The measurements presented here do not include warm-up time.

The comparison of the control dependencies query is shown in Figure 3.19. The plot shows that the performance is quite comparable. On average, the *Shy* implementation of the query seems a little slower. This is probably caused by the extensive use of interfaces in the *Shy* framework, whereas the AST-based implementation only uses abstract and concrete classes. For transformations the performance difference is slightly more pronounced. Figure 3.20 shows the performance comparison of

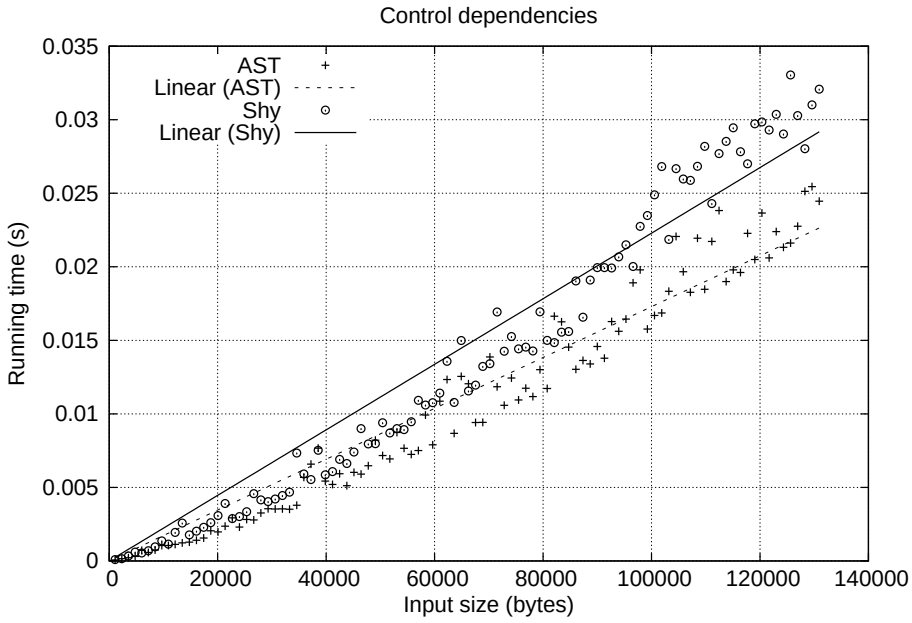

Figure 3.19: Performance comparison of control dependencies query.

the inline conditions transformation. The greater difference can be explained by the fact that creating a new structure in a *Shy* transformation involves dynamically dispatched method calls instead of statically bound constructor calls.

3.9.4 *Shy* vs Vanilla Regarding Code Size

The percentages shown in Table 3.1 illustrate the structure-shyness of queries and transformations implemented using *Shy*. Table 3.2 shows the absolute number of source lines of code (SLOC, lines of code without counting empty lines and comments). For the vanilla AST-based implementation we only show the total SLOC count, since all operations are scattered over the respective AST classes. For the *Shy* implementation, each query and operation is realized a separate class or interface, extending the interfaces generated by *Shy*. In total, one can observe that the *Shy*-based implementation requires less than half of the number of lines of code required in the vanilla implementation. Note also that *Shy* supports a much more modular design, where both the AST data type and the set of operations can be extended without having to modify existing code.

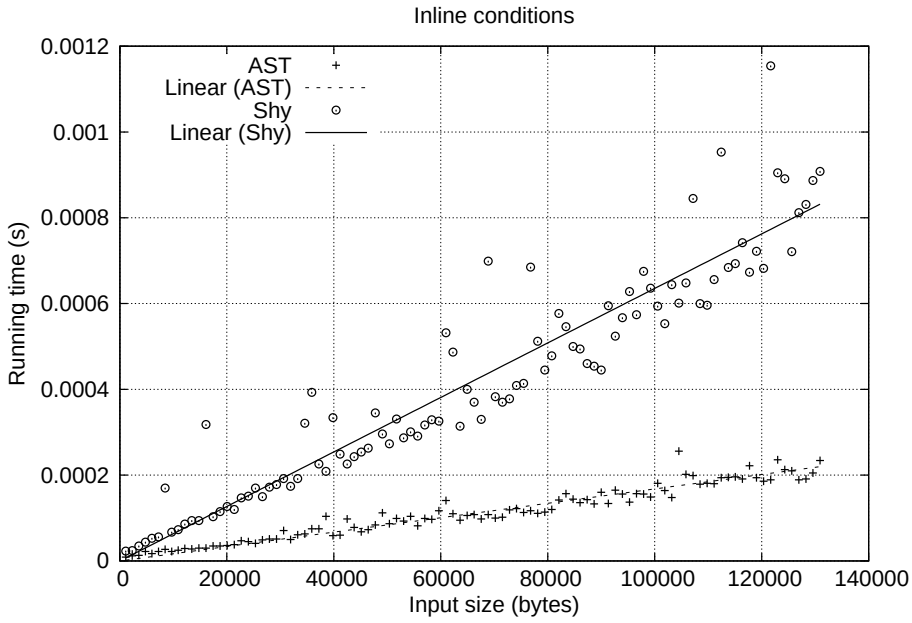

Figure 3.20: Performance comparison of inline conditions transformation.

Table 3.2: Source Lines of Code (SLOC) statistics: *Shy* implementation vs Vanilla AST implementation.

Component	SLOC
Object Algebra interfaces	71
Collect variables	10
Data dependencies	30
Control dependencies	58
Type environment	15
Rename variable	27
Inline conditions	48
Desugar "unless"	10
Desugar "repeat"	35
Total Shy-based operations	304
AST-based implementation	661

3.10 Summary

This chapter showed how various types of default traversals for complex structures can be automatically provided by *Shy*. *Shy* traversals are written directly in Java and are type-safe, extensible and

separately compilable. There has always been a tension between the correctness guarantees of static typing, and the flexibility of untyped/dynamically-typed approaches. *Shy* shows that even in type systems like Java's, it is possible to get considerable flexibility and adaptability for the problem of boilerplate code in traversals of complex structures, without giving up modular static typing.

Chapter 4

Type-Safe Modular Parsing

In most related work on the Expression Problem [Wadler, 1998] and its extended areas, people are usually concerned about *semantic modularity* on consumer operations that *traverse* or *process* data structures. In contrast, there is surprisingly little focus on producers that *build* extensible ASTs with type-safe modularity, of which parsing is a representative. Even for the tip of the iceberg, most existing parsing techniques merely focus on *syntactic modularization* of grammars before code generation.

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

Algorithmic Challenges A first challenge was to do with the parsing algorithms themselves, since they were usually not designed with extensibility in mind. The most widely used tools for parsing are parser generators [Gouseti et al., 2014; Grimm, 2006; Parr and Quong, 1995; Schwerdfeger and Van Wyk, 2009a; Viera et al., 2012; Warth et al., 2016b], but they mostly require full information about the grammar to generate parsing code. Moreover, actions associated with grammar productions are typically only type-checked after the parser has been generated. Both problems go against our goals of semantic modularity.

An alternative to parser generators are *parser combinators* [Burge, 1975; Wadler, 1985]. At a first look, parser combinators seem very suitable for our purpose. Each parser combinator is represented by a piece of code directly in the programming language. Thus, in a statically typed programming language, such code is statically type-checked. However many techniques regularly employed by parser combinators cause difficulties in a modular setting. In particular, many parser combinator approaches (including Parsec [Leijen and Meijer, 2001]) routinely use *left-recursion elimination*, *priority-based matching*, and avoid *backtracking* as much as possible. All of these are problematic in a modular setting as illustrated in Section 4.1.1.

To address such algorithmic challenges, we propose a methodology for implementing modular parsers built on top of an existing Packrat [Ford, 2002] parsing library for Scala [et al., 2004]. Such a library directly supports left-recursion, memoization, and a longest-match composition operator. Some examples will be shown in Section 4.1.2.

Typing and Reusability Challenges The second class of challenges was problems related to modularity, reusability and typing of parsing code. An immediate concern is how to extend a parser for an existing language or, more generally, how to compose parsing code for two languages. It turns out that OO mechanisms that provide some form of multiple inheritance, such as traits/mixins [Bracha and Cook, 1990; Schärli et al., 2003], are very handy for this purpose. Essentially, traits/mixins can act as modules for the parsing code of different languages. This enables an approach where ASTs can be modelled using standard OO techniques such as the Composite pattern, while retaining the possibility of adding new language constructs. Section 4.2 gives the details of this approach. Our ultimate goal is to allow for full extensibility: it should be possible to modularly add not only new language constructs, but also new operations. To accomplish this goal one final tweak on our technique is to employ Object Algebras to allow fully extensible ASTs. Thus a combination of Packrat parsing, multiple inheritance and Object Algebras enables a solution for semantically modular parsing. Section 4.3 gives the details of the complete approach. To evaluate our approach we conduct a case study based on the "Types and Programming Languages" (TAPL) [Pierce, 2002] interpreters. The case study shows that our approach is effective at reusing parsing code from existing interpreters, and the total parsing code is 69% shorter than an existing code base using non-modular parsing code[1].

Overall, this chapter presents code in Scala, due to the use of Scala Packrat parsing library. Section 4.1 studies the main algorithmic challenges in achieving modular parsing, to motivate Packrat parsing as an eligible technique. Section 4.2 demonstrates how OO inheritance tackles the recursion problem in defining extensible parsers. Section 4.3 discusses full extensibility from Object Algebras as the representation of data structures. Section 4.4 explores more interesting features from the aspect of implementation. Finally Section 4.5 presents our case study based on TAPL interpreters, and shows significant reuse in parsing code by our technique.

4.1 Packrat Parsing for Modularity

This section discusses the algorithmic challenges introduced by modular parsing and argues that Packrat parser combinators [Ford, 2002] are suitable to address them. The algorithmic challenges are important because they rule out various common techniques used by non-modular code using parser combinators. To avoid pitfalls related to those algorithmic challenges, we propose the following methodology:

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

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

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

4.1.1 Algorithmic Challenges of Modularity

For the goal of modular parsing, parser combinators seem suitable because they are naturally modular for parser composition, but also they ensure type safety. Unfortunately many parser combinators have important limitations. In particular, several parser combinators including the famous Parsec [Leijen and Meijer, 2001] library, require programmers to manually do *left-recursion elimination*, *longest match composition*, and require significant amounts of *backtracking*. All of those are problematic in a modular setting.

Left-Recursion Elimination The top-down, recursive descent parsing strategy adopted by those parser combinator libraries cannot support left-recursive grammars directly. For instance, we start with a simple arithmetic language containing only integers and subtractions, with the following concrete syntax:

```
<expr> ::= <int>
         |  <expr> '-' <int>
```

And part of the parsing code with Parsec would be:

```
parseExpr = parseSub <|> parseInt

parseSub = do
  e <- parseExpr ...
```

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

```
<expr>  ::= <int> <expr'>

<expr'> ::= <empty>
          |  '-' <int> <expr'>
```

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

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

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

```
<expr> ::= <int>
         |  <int> '+' <expr>
```

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

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

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

```
<stmt> ::= 'import' <ident> 'from' <ident>
         |  ...
         |  'import' <ident> 'as' <ident>
```

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

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

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

4.1.2 A Solution: Packrat Parsing

Fortunately, some more advanced parsing techniques such as Packrat parsing [Ford, 2002] have been developed to address limitations of simple parser combinators. Section 2.5 has introduced the necessary background about Packrat parser combinators with a concrete example. Packrat parsing uses memoization to record the result of applying each parser at each position of the input, so that repeated computation is eliminated. Moreover, it supports both direct left-recursion and (in theory) indirect left-recursion [Warth et al., 2008]. Additionally, the standard Scala Packrat parsing library [Moors et al., 2008] provides a number of parser combinators, including the longest match alternative combinator. All of these properties are very suitable for modularity, thus we decided to use Packrat parsers as the underlying parsing technique for modular parsing.

For more concise demonstration, we assume that all Scala code in the rest of this chapter is in the object `Code` shown in Figure 4.1. It extends `StandardTokenParsers` and `PackratParsers` from the Scala parser combinator library. Furthermore, we use `Parser` as a type synonym for `PackratParser` and a generic `parse` function for testing.

As a result, we simplify the `ExpParser` from Section 2.5 into the following trait that only contains the actual parsing code:

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

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

Figure 4.1: Helper object for code demonstration in this chapter.

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

4.2 OO AST Parsing with Multiple Inheritance

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

Part of the modular parsing problem is how to obtain an extensible parser. It is natural to make use of OO ASTs because adding new data constructs is cheap for them. Hence we have used OO traits and inheritance to represent the AST in Section 2.5. Furthermore, we would like to write extensible parsing code on extensions of a grammar. That is to say, new extensions would not require modifying the existing code, and we can even reuse the old code.

To illustrate such extensibility, we continue with the arithmetic language from Section 2.5, and introduce variables as a new case. It is easy to extend the corresponding OO AST together with its parser in a modular way:

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

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

`pExpr` with `"pExpr ||| pVar"`. Yet modifying existing code breaks semantic modularity.

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

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

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

Independent Extensibility A nice feature of Scala is its support for the linearized-style multiple inheritance on traits [et al., 2004]. This can be very helpful when composing several languages, and to achieve independent extensibility [Odersky and Zenger, 2005a]. Suppose now we want to compose the parsers for expressions from pre-defined languages `LanguageA` and `LanguageB` using alternative. The new parser can be built by inheriting both parsers at the same time:

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

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

Conflicts and/or Ambiguity In a modular setting, conflicts and ambiguity could be introduced to the grammar. In that case, the help parser combinators can offer is quite restricted. Yet users can override those problematic methods to resolve such conflicts, and rely on dynamic dispatch. We will discuss it in Section 4.4.2.

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

4.3 Full Extensibility with Object Algebras

The inheritance-based approach allows building extensible parsers, based on an OO class hierarchy. Nevertheless, the addition of new operations over ASTs is problematic using traditional OO ASTs. In this section, we show how to support both forms of extensibility on ASTs (easy addition of language constructs, and easy addition of operations) using Object Algebras [Oliveira and Cook, 2012a].

4.3.1 Problem with Traditional OO ASTs

Section 2.5 uses the conventional *Interpreter* pattern to represent data, and this again refers to the Expression Problem [Wadler, 1998]. It has been discussed in Section 2.2.1 that such a pattern makes it difficult to add new operations. To modularly add an operation like collecting free variables, one attempt would be extending `Expr` with the new operation to obtain a new abstract type for ASTs:

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

Firstly, such an approach is known to be problematic in terms of type-safety (but see recent work by Wang and Oliveira [2016], which shows a technique that is type-safe in many cases). More importantly, a second problem is that even if that approach would work, the parsing code in `VarParser` is no longer reusable! The types `Expr, Lit, Add`, and so on, are all old types without the free variables operation. To match the new ASTs, we have to substitute `NewExpr` for `Expr` (the same for `Lit`, `Add`, ...). This requires either code modification or type casts. The goal of semantic modularity motivates us to adopt a different approach for building ASTs.

4.3.2 Parsing with Object Algebras

Due to the object-oriented context, Scala provides a nice platform for realizing Object Algebras. In that case, we can separate the definition of data variants from operations, while retaining extensibility in both dimensions.

Object Algebras for Both Extensibility Firstly, the AST is refactored with an Object Algebra interface:

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

Next comes the pretty-printing by instantiating the above trait into an algebra:

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

Subsequently, variables can be modularly added without modifying existing code:

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

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

Figure 4.2: Pattern of modular parsing using Object Algebras.

```
}

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

Nevertheless, object as the result of parsing, have a different representation in Object Algebras, namely a function `Alg[E] => E`. When Object Algebras are used to build ASTs, an Object Algebra containing the constructor/factory methods has to be used by the parsing function. Thus, a first attempt at defining the parser for the small arithmetic language is:

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

Such a parser looks fine, but it is not extensible. For example, we have demonstrated in Section 4.2 that method overriding is essential to update `pExpr` for an extended syntax. However, trying to do a similar method overriding for `pExpr` would require a type `VarAlg[E] => Parser[E]`, which is a *supertype* of the old type `Alg[E] => Parser[E]`, since the *extended* Object Algebra interface appears in *contravariant* position. This violates overriding in Scala.

A Solution A solution to this problem is to declare a field of Object Algebra interface in the parser. Figure 4.2 shows the code of true modular parser, whose methods can be overridden for future extension.

That is precisely the pattern that we advocate for modular parsing. One important remark is we introduce `pE` for recursive calls. The reason why we use it as an extra and seemingly redundant field, is due to a subtle issue caused by Scala language and its parser combinator library. There is a restriction of **`super`** keyword in Scala that **`super`** can only use methods defined by keyword **`def`**, but

cannot access fields defined by **val**, while the parser combinator library suggests using **val** to define parsers, especially for left-recursive ones. Our workaround is that we use different synonyms for pE in different traits, so that we can directly distinguish them by names without using **super**.

EXTENSIONS Now let's try on the variables extension:

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

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

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

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

Note that, similar to the approach in Section 4.2, independent extensibility is also supported via multiple trait inheritance. Since it is achieved using essentially the same technique as in Section 4.2, we omit the code here.

4.4 More Features

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

4.4.1 Parsing Multi-Sorted Syntax

Using Object Algebras, it is easy to model multi-sorted languages. If the syntax has multiple sorts, we can distinguish them by different type parameters. For instance, we extend the expression language from the end of Section 4.3, with a primitive type int type and typed lambda abstractions:

```
<type> ::= 'int'

<expr> ::= ...
        |  '\' <ident> ':' <type> '.' <expr>
```

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

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

We use two type parameters `E` and `T` for expressions and types. The type system guarantees that invalid terms such as `int + int` will be rejected. Besides lexing, the trait `LamOAParser` also introduces parsers for types, and the new case for expressions. We use `pTypedLamT` and `pTypedLamE` as copies of current `pT` and `pE`, due to the issue with **`super`** in Scala (see discussion in Section 4.3.2). `pT` and `pE` are used for recursion.

4.4.2 Overriding Existing Rules

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

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

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

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

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

4.4.3 Language Components

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

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

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

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

We take the example in Section 4.3.2 again. It can be defined as a language component `VarExpr`. For space reasons we omit some detailed code.

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

For the extension of types and lambda abstractions in Section 4.4.1, instead of inheriting from the previous language directly, we can define it as another independent language component `TypedLam`.

```
object TypedLam {
  trait Alg[E, T] {    // Abstract syntax
    def intT(): T
    ...
  }
  trait Parse[E, T] { ... } // Parser
```

```
    trait Print extends Alg[String, String] {
      ... // Pretty-printer
    }
  }
```

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

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

The only drawback is that the glue code of composition appears to be boilerplate. As shown above, we are combining ASTs, parsers and pretty-printers of `VarExpr` and `TypedLam` respectively. Such a pattern refers to *family polymorphism* [Ernst, 2001] which is unfortunately not fully supported in Scala, since nested classes/traits have to be manually composed.

4.4.4 Alternative Techniques

Our prototype uses Packrat parsing as the underlying parsing technique, OO inheritance for composing and extending parsers, and Object Algebras for parsing extensible ASTs. Yet such a framework is itself flexible and modular, because those techniques can have alternatives. For example, as we mentioned before, any parsing library that resolves the algorithmic challenges in modular parsing can work well. Regarding OO inheritance for the extensibility, an alternative approach, called *open recursion* [Cook, 1989] can be used in other languages, by introducing explicit "self-reference" parameters for the recursion. Furthermore, besides Object Algebras, *Data types à la carte* (DTC) [Swierstra, 2008] and the Cake pattern [Odersky and Zenger, 2005a] also support extensible data structures. For the goal of modular parsing a custom combination of those alternatives can be adopted.

4.5 Case Study

To demonstrate the utility of our modular parsing approach, we implemented parsers for the first 18 calculi [1] from the *Types and Programming Languages* (TAPL) [Pierce, 2002] book. We compared our implementation with a non-modular implementation available online, which is also written in Scala and uses the same Packrat parsing library. We counted source lines of code (SLOC) and measured execution time for both implementations. The result suggests that our implementation saves 69% code comparing with that non-modular one, but there is a 43% slowdown due to code modularity.

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

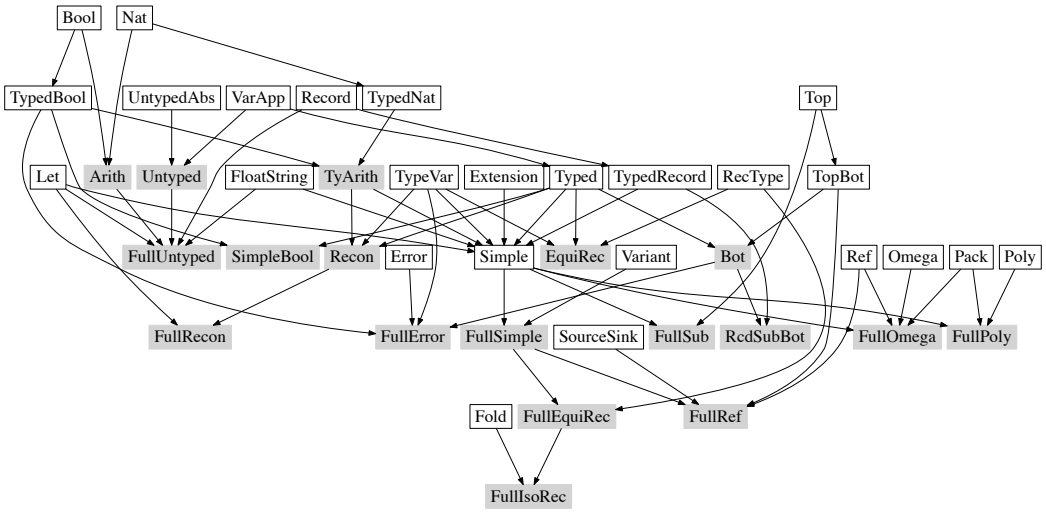

Figure 4.3: Dependency graph of all calculi and components. Grey boxes are calculi; white boxes are components.

4.5.1 Implementation

TAPL introduces several calculi from simple to complex, by gradually adding new features to syntax. These calculi are suitable for our case study for mainly two reasons. Firstly, they capture many of the language features required in realistic programming languages, such as lambdas, records and polymorphism. Secondly, the evolution of calculi in the book reveals the advantages of modular representation of abstract syntax and modular parsing, which is the key functionality of our approach. By extracting common components from those calculi and reusing them, we obtain considerably code reuse as shown later.

We extract reusable components from all the calculi using the pattern in Section 4.4.3. Each component, which may contain several syntactical structures, represents a certain feature. They are combined together as needed to build a calculus. For example, the calculus `Untyped` in our case study, representing the famous untyped lambda calculus, consists of component `VarApp` (for variables and applications) and component `UntypedAbs` (for untyped lambdas).

Figure 4.3 shows the dependency of all the components and calculi in our case study. Grey boxes are calculi and white boxes are components. An arrow starting from box A to box B denotes that B includes and thus reuses A.

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

Table 4.1: Comparison of SLOC and execution time.

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

4.5.2 Comparison

We compared our implementation (named $\texttt{Mod}_{\texttt{OA}}$) with an implementation available online[1] (named `NonMod`). `NonMod` is suitable for comparison, because it is also written in Scala using the same parser combinator library. `NonMod` implements parsers 18 calculi in TAPL in a non-modular way. Thus `NonMod` is not able to reuse existing code when those calculi share common features. $\texttt{Mod}_{\texttt{OA}}$ implements the same 18 calculi, but reuse is possible due to modularity.

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

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

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

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

Comparison Results Table 4.1 shows results of the comparison. Let us only check $\texttt{Mod}_{\texttt{OA}}$ and `NonMod` for now. The overall result is that 69.2% of code is reduced using our approach, and our implementation is 42.7% slower.

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

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

The right part of Table 4.1 suggests that the difference of running time between using Object Algebras and class is little, roughly 1%. The use of longest match combinator slows the performance by 7%. The main reason of slower execution time may be the overall structure of the modular parsing approach, because we indeed have more intermediate function calls and method overriding. However, it is worth mentioning that because of the memoization technique of Packrat parsers, we are only constant times slower, the algorithmic complexity is still the same. Since the slowdown seems to be caused by extra method dispatching, in future work we wish to investigate techniques like partial evaluation or meta-programming to eliminate such cost. The work by Béguet and Jonnalagedda [2014] is an interesting starting point.

4.6 Summary

This chapter presents a solution for type-safe modular parsing. The solution not only enables parsers to evolve together with the abstract syntax, but also allows parsing code to be modularly type-checked and separately compiled. We identify the algorithmic challenges of building modular parsers, and use standard OO techniques including inheritance and overriding for our goal. However, the extensibility issue of traditional OO ASTs motivates us to adopt Object Algebras for full extensibility and more useful features. Then language feature abstraction further enhances code reuse and modularity.

Chapter 5

Modular Unfolds: Seeing the Trees in the Product Forest

The type-safe modular parsing approach, presented in Chapter 4, implies a general way to modularize *producer* operations using inheritance (or open recursions). However, this chapter gives a different perspective of modularizing producers in functional programming, by modularizing *coalgebras*.

Based on AoP [Bird and de Moor, 1997] and DTC [Swierstra, 2008], it is known that the modularization of *consumer* operations can be achieved by modularizing *algebras*. Thus the duality between algebras and coalgebras motivates our exploration in modularizing coalgebras. However, composing coalgebras is not as obvious as composing algebras, since the natural, generic composition operator produces a structure with products (instead of sums) at the top-level nodes. Taking the fixpoint of such structure results in a so called *product forest*, instead of the more familiar and expected fixpoint of sums-of-products. One possible interpretation of a product forest is as a collection of possible sums-of-product structures that can be generated from the coalgebras. This chapter will show that, to recover a particular tree (i.e. a sum-of-product structure), we can define a selection function that extracts the desired tree(s) from the product forest. There are many possible ways to define such a selection function. For example, one can randomly choose between the various products, or simply generate a list of all possible trees. We show how to build tree structures as a second step after the construction of a product forest, by defining corresponding selection functions.

In order to eliminate the construction of product forests, and directly build a sums-of-product structure, specialized composition combinators can be derived for coalgebras. These combinators can then be used instead of the generic composition operator to compose coalgebras, avoiding the intermediate data structure. The correctness of such deforestation [Wadler 1988] process is validated by corresponding fusion theorems. Selection functions become much more interesting in the presence of effects, which motivates the generalization of our work to monadic coalgebras.

As a consequence, we provide *SCCL*, a Haskell library to present some specialized combinators for practical use. It includes some basic combinators that reveal general composition strategies, and a few more specialized combinators for several applications: *modular random generators*, *modular small-step semantics* and *modular monadic parsing*. Consequently, with *SCCL* we are able to achieve:

- *Modularity of AST and producers:* Data variants are modularized with functors, and producers can be defined as coalgebras, composable via the general coalgebra combinator.

- *Untangling of strategies:* Using coalgebras and the generic unfold, production strategies get untangled with composition strategies.
- *Encapsulation and reusability of composition strategies:* Composition strategies can be implemented as natural transformations, which further derive specialized coalgebra combinators with genericity on the carrier type.
- *Reusability of producer coalgebras:* With monad transformers, coalgebras become reusable components for different composition strategies.
- *Code conciseness:* The specialized combinators free programmers from duplicating boilerplate composition code, and hence simplify client code significantly.

This chapter is organized as follows. Section 5.1 motivates the modularity issue by an example of random generators, and gives an overview of the *SCCL* library. Section 5.2 shows the generic coalgebra combinator and illustrates the resulting product forests from unfolding. Section 5.3 shows that the more desired sum-of-products trees can be derived from product forests using natural transformations. Section 5.4 further generalizes the theory to monadic variants. And finally Section 5.5 presents the implementation of *SCCL*, including the basic combinators and the specialized combinators for the three applications.

5.1 Overview

This section gives an overview of *SCCL* (Specialized Coalgebras Combinator Library). We start with QuickCheck generators as a motivating example, showing the problem of code modularity and reusability. Then we present how *SCCL* modularizes random generators with coalgebras and specialized combinators, and further present an overview of *SCCL* combinators.

5.1.1 A Motivating Example: QuickCheck Generators

QuickCheck [Claessen and Hughes, 2000] is a Haskell library well-known for random testing of programs. Programmers define some properties that they expect their programs to satisfy, in a logical form with boolean functions, operators and quantifiers. Then QuickCheck automatically generates a number of values or structures to test if the properties hold.

Besides primitive types, QuickCheck also allows generators to be implemented for custom datatypes. Our motivating example is a simple language that supports numeric and boolean literals, addition, and equality, defined as follows by a recursive datatype:

```
data Exp = Lit Int | BoolLit Bool | Add Exp Exp | Equal Exp Exp
```

A property can be, for example, the type preservation of evaluating such expressions, namely if an expression is well-typed, then performing evaluation rules on the expression does not change its type. However, the effectiveness of random testing depends highly on the implementation of generators. Often, a "good" generator is at least expected to cover all the constructs, and ensure some randomness. In related work [Claessen et al., 2014; Duregård et al., 2012; Fetscher et al., 2015; Grygiel and Lescanne, 2013; Pałka et al., 2011], people have different opinions on how a good generator looks like. Below we give three possible implementations.

```
instance Arbitrary Exp where
  arbitrary = sized gen1
    where gen1 :: Int -> Gen Exp
          gen1 n | n <= 0    = frequency [(1, gLit),   (1, gBool)]
                 | otherwise = frequency [(1, gLit),   (1, gBool),
                                          (1, gAdd n), (1, gEqual n)]
          gLit  = liftM Lit $ choose (0, 100)
          gBool = liftM BoolLit $ choose (False, True)
          gAdd   n = do eA <- gen1 (n - 1)
                        eB <- gen1 (n - 1)
                        return (Add eA eB)
          gEqual n = do eA <- gen1 (n - 1)
                        eB <- gen1 (n - 1)
                        return (Equal eA eB)
```

Figure 5.1: Uniformly generating expressions on a specific size.

I. Generating expressions up to a specific size, with a uniform distribution. The uniform distribution is arguably a most common strategy of random generation [Claessen et al., 2014; Duregård et al., 2012; Fetscher et al., 2015; Grygiel and Lescanne, 2013]. Furthermore, generators are usually restricted with an input size, though such a size may have different interpretations. Figure 5.1 shows a possible implementation, where `n` stands for the *depth size*; here it represents the upper bound of the *height* of the tree to generate. The height of a tree is precisely the maximum distance between the root and a leaf. When `n` is non-positive, only leaf nodes can be generated: a number (between 0 and 100) or a boolean value, each with 0.5 probability. When `n` is greater than 0, all four cases share the same probability, and in `Add` or `Equal`, the sub-expressions are recursively built from a smaller size `n - 1`. The uniformity here is not for all functors, but for the candidates that satisfy certain conditions.

II. Generating well-typed expressions up to a specific size, with a uniform distribution. When testing compilers or interpreters, people sometimes would like the generator to generate only well-typed expressions [Claessen et al., 2014; Fetscher et al., 2015; Grygiel and Lescanne, 2013; Pałka et al., 2011]. Figure 5.2 shows one implementation for `Exp`, considering that the language only has two types: integer and boolean. Compared with `gen1`, `gen2` takes an additional parameter for the expected type, and it is initialized as `Any`.

III. Generating expressions up to a specific size, with a dynamic distribution. Inspired by [Fetscher et al., 2015], a random generation can also follow a dynamic distribution, where constructors with more branches (like `Add` and `Equal`) tend to be selected at the beginning of construction for expansion, and constructors with fewer branches (like `Lit` and `BoolLit`) are more likely to be selected when the tree goes deeper for quick convergence. In Figure 5.3, `gen3` takes two arguments, where `n` represents the current size, and `m` records the original input (maximum size). When `n` becomes smaller during tree generation, the weights are changed dynamically.

Overall, those strategies can be chosen for different situations flexibly. Unfortunately, by comparing

```
instance Arbitrary Exp where
  arbitrary = sized $ gen2 Any
    where gen2 :: Type -> Int -> Gen Exp
          gen2 TInt  n | n <= 0    = gLit
                       | otherwise = frequency [(1, gLit),   (1, gAdd n)]
          gen2 TBool n | n <= 0    = gBool
                       | otherwise = frequency [(1, gBool),  (1, gEqual n)]
          gen2 Any   n | n <= 0    = frequency [(1, gLit),   (1, gBool)]
                       | otherwise = frequency [(1, gLit),   (1, gBool),
                                                (1, gAdd n), (1, gEqual n)]
          gLit  = liftM Lit $ choose (0, 100)
          gBool = liftM BoolLit $ choose (False, True)
          gAdd   n = do eA <- gen2 TInt (n - 1)
                        eB <- gen2 TInt (n - 1)
                        return (Add eA eB)
          gEqual n = do t  <- elements [TInt, TBool]
                        eA <- gen2 t (n - 1)
                        eB <- gen2 t (n - 1)
                        return (Equal eA eB)
```

Figure 5.2: Uniformly generating well-typed expressions on a specific size.

the three pieces of code, we have noticed two critical issues in terms of modularity and code reuse: *entanglement of strategies*, and *non-extensibility of generators*.

ENTANGLEMENT OF STRATEGIES We have noticed that there are two orthogonal dimensions of strategies. The first one is the strategy of generation, which represents the behavior of each individual producer (`gLit`, `gBool`, `gAdd` and `gEqual`). For this, I and III are basically the same; they both generate well-formed expressions up to a certain size. However, II is different from I, since it takes the additional type information to generate expressions that always type-check. The second dimension of strategy refers to how those individual generators are composed (i.e. the composition/distribution strategy). For this, I and II both adopt the uniform distribution, by setting equal weights in `frequency`. Whereas III uses a dynamic distribution algorithm to generate expressions of a similar shape. It is unfortunate to see that these two dimensions of strategies entangled in the code without reusability, as a consequence, a lot of code has to be duplicated.

NON-EXTENSIBILITY OF GENERATION Another critical issue is that the above code lacks extensibility and modularity. The language constructs are defined in `Exp`, a non-extensible abstract syntax tree. Furthermore, the generators are defined as a family of mutually recursive functions; they are non-modular because such recursion is closed. Nowadays, with the rise of modular compilers and interpreters [Ekman and Hedin, 2007; Inostroza and Storm, 2015; Liang et al., 1995; Nystrom et al., 2003], there is also need for modularization in producer operations, including this random generation. The non-extensibility of the above code, however, results in significant code duplication.

Suppose we extend `Exp` with multiplications. The AST might be redefined as follows:

```
instance Arbitrary Exp where
  arbitrary = getSize >>= \n -> gen3 n n
    where gen3 :: Int -> Int -> Gen Exp
          gen3 n m = frequency [(n, gAdd n m),
                                (n, gEqual n m),
                                (m + 1 - n, gLit),
                                (m + 1 - n, gBool)]
          gLit  = liftM Lit $ choose (0, 100)
          gBool = liftM BoolLit $ choose (False, True)
          gAdd n m   = do eA <- gen3 (n - 1) m
                          eB <- gen3 (n - 1) m
                          return (Add eA eB)
          gEqual n m = do eA <- gen3 (n - 1) m
                          eB <- gen3 (n - 1) m
                          return (Equal eA eB)
```

Figure 5.3: Generating expressions in a dynamic distribution.

```
data Exp = Lit Int | BoolLit Bool | Add Exp Exp | Equal Exp Exp
                   | Mul Exp Exp -- extension: multiplication
```

Figure 5.4 gives a modified implementation of generator I for this AST. The code is mostly the same as Figure 5.1, except that `gMul` is added. Clearly modifying existing code is not expected, but instead we have to duplicate much code, for neither the AST nor the recursive functions are open to extensions. Additionally, programmers have to take responsibility for carefully rearranging and checking the distribution with extensions, which is especially error-prone when the distribution strategy gets complicated. Hence our wish is to modularize generator code, and also to free programmers from dealing with the strategies manually, resulting in lightweight and type-safe composition.

5.1.2 Solution in *SCCL* for Random Generation

The aforementioned issues have motivated our exploration on modular unfolds, and the implementation of our *SCCL* library. In *SCCL*, producer functions (such as generators) are defined as *coalgebras*, composed by *specialized combinators*, and finally fed to a generic *unfold* to build data structures.

Coalgebras and Unfold for Producers From DTC [Swierstra, 2008] and Section 2.3.3, we know that *functors* and their *co-products* can represent extensible data structures in Haskell. The previous `Exp` type can be separated into four functors:

```
data LitF x   = Lit Int
data BoolF x  = BoolLit Bool
data AddF x   = Add x x
data EqualF x = Equal x x
```

By default we make all those functors instances of `Functor`. Here `x` is the *carrier* to capture recursive structures. As the second step, taking the fixpoint of a functor gives us the real data type for abstract

```
instance Arbitrary Exp where
  arbitrary = sized gen1
    where gen1 :: Int -> Gen Exp
          gen1 n | n <= 0    = frequency [(1, gLit),   (1, gBool)]
                 | otherwise = frequency [(1, gLit),   (1, gBool),
                                          (1, gAdd n), (1, gEqual n),
                                          (1, gMul n)]
          gLit  = liftM Lit $ choose (0, 100)
          gBool = liftM BoolLit $ choose (False, True)
          gAdd   n = do eA <- gen1 (n - 1)
                        eB <- gen1 (n - 1)
                        return (Add eA eB)
          gEqual n = do eA <- gen1 (n - 1)
                        eB <- gen1 (n - 1)
                        return (Equal eA eB)
          gMul   n = do eA <- gen1 (n - 1)
                        eB <- gen1 (n - 1)
                        return (Mul eA eB)
```

Figure 5.4: Uniformly generating expressions on a specific size (added multiplications).

syntax. In this example, `Fix (LitF ⊕ BoolF ⊕ AddF ⊕ EqualF)` is isomorphic `Exp`.

Since generators build data structures, this assumes that they can be captured by (top-down) unfolds. With this assumption, a generator should be represented by a (monadic) coalgebra. Using coalgebras and the generic unfold pattern gives us the following benefits:

- *The AST and its generator become extensible.* Language constructs are now encouraged to be defined by individual functors, and composed with sums. This allows extensibility on the AST; and meanwhile, generators (coalgebras) become by modular components. This modularization makes the foundation for code reuse.

- *Producers are separated from the generic recursion pattern.* The use of coalgebras and unfold gives an additional benefit: the real production process is separated from the recursion pattern. In Figure 5.1, `gen1` is manually invoked several times, and so are `gen2` and `gen3`. Yet such boilerplate can be captured by *unfold*, a generic recursion for building data structures. Making use of generic recursion schemes, instead of ad-hoc recursive functions, gets programs easier to reason about and optimize.

- *The two dimensions of strategies get untangled automatically.* Coalgebras represent the production strategies; and the composition of coalgebras, though not as easy as composing algebras, can be achieved by specialized combinators which reveal certain composition/distribution strategies. By adding a bit of abstraction, the two kinds of strategies get untangled, and can further form various combinations with flexibility and reusability. Meanwhile, specialized combinators make the composition more concise and less error-prone in client code.

GENERATORS AS REUSABLE COALGEBRAS Recall that a monadic coalgebra has the following type:

```
type CoAlgM m f a = a -> m (f a)
```

At this point, one may want to instantiate `m` with `Gen`. However, in order to define coalgebras as reusable components, we use *monad transformers* [Liang et al., 1995] to abstract the monad. Hence the generator in Figure 5.1 is refactored into the following coalgebras:

```
gLitF :: MonadRand m => CoAlgM m LitF Int
gLitF _ = liftM Lit $ choose (0, 100)

gBoolF :: MonadRand m => CoAlgM m BoolF Int
gBoolF _ = liftM BoolLit $ choose (False, True)

gAddF :: (MonadMaybe m, MonadRand m) => CoAlgM m AddF Int
gAddF n | n <= 0    = none
        | otherwise = return $ Add (n - 1) (n - 1)

gEqualF :: (MonadMaybe m, MonadRand m) => CoAlgM m EqualF Int
gEqualF n | n <= 0    = none
          | otherwise = return $ Equal (n - 1) (n - 1)
```

where

```
none :: MonadMaybe m => m a
```

stands for a failure, and

```
choose :: MonadRand m => Random a => (a, a) -> m a
```

is a generalization of the `choose` in QuickCheck for random selection in a specific range. Those coalgebras are polymorphic on the monad, and all have the same carrier type, namely `Int` for depth size.

COMPOSING COALGEBRAS *SCCL* provides a number of specialized coalgebra combinators. For a uniform distribution, the monad is instantiated to

```
type Weighted = MaybeT (WeightT (RandT Identity))
```

where `MaybeT` corresponds to `MonadMaybe` for encoding failure. `WeightT`, defined in *SCCL*, models a weighted distribution by maintaining a list of weights in state; it is similar to `frequency`, but designed for coalgebra composition in a binary form. `RandT`, also from *SCCL*, is like a generalization of `"QuickCheck.Gen"` using monad transformers; some auxiliary functions, including `choose`, are captured by `MonadRand`.

SCCL defines the combinator $|*|_w$ specialized for `Weighted`, with the following signature:

```
(|*|_w) :: Cardinality g =>
    CoAlgM Weighted f a -> CoAlgM Weighted g a -> CoAlgM Weighted (f ⊕ g) a
```

It realizes such a composition strategy: to perform a *weighted random distribution* over *successful* generators for each construction. The constraint `Cardinality g` is used to check the border in a chained composition; it will be introduced later. The generator can be modularly constructed and tested in the `IO` environment:

```
generateW :: (Cardinality f, Traversable f) =>
    [Double] -> a -> CoAlgM Weighted f a -> IO (Fix f)
generateW = ...

testW :: Int -> IO (Fix (LitF ⊕ BoolF ⊕ AddF ⊕ EqualF))
testW n = generateW [1, 1, 1, 1] n gen
  where gen = gLitF |*|w gBoolF |*|w gAddF |*|w gEqualF
```

Here `generateW` is an auxiliary function; it takes three arguments: the list of weights, the input, and the (composed) coalgebra. It applies the generic monadic unfold to the coalgebra, and then evaluates the monad under **`IO`**. Finally `testW` has the same functionality as Figure 5.1, since all weights are set to 1.

Besides $|*|_w$, *SCCL* provides another combinator for a dynamic distribution (different from III). When the monad is instantiated to `Dynamic`:

```
type MaxArity = Int
type MaxDepth = Int
type ReadEnv = (MaxArity, MaxDepth)
type Acc = Double
type Bound = Int
type StateEnv = (Acc, Bound)

type Dynamic = MaybeT (StateT StateEnv (ReaderT ReadEnv (RandT Identity)))
```

The specialized combinator

```
(|*|d) :: (Cardinality g, Arity f, Arity g, Derive a Int) =>
    CoAlgM Dynamic f a -> CoAlgM Dynamic g a -> CoAlgM Dynamic (f ⊕ g) a
```

samples a binomial distribution over functors, based on their *arities* and the depth of tree generation. We use the term "arity" to represent the number of sub-nodes/branches in a construct. In the above example, `LitF` and `BoolF` have arity 0, while `Add` and `Equal` have arity 2. Similar to Figure 5.3, this strategy arranges more probability to high-arity constructs near the root, and more preference on low-arity constructs when the generation goes deeper. The carrier type is still abstract, but the constraint `"Derive a Int"` requires that it includes an integer for observing the depth of generation. By the binomial distribution, the probabilities are changed more smoothly.

Now the generators can instantly be composed by $|*|_d$:

```
gen' = gLitF |*|d gBoolF |*|d gAddF |*|d gEqualF
```

before fed to `unfoldM`. This reflects the reusability of coalgebras with different composition strategies.

REUSABILITY OF COMBINATORS Since the combinators have abstracted the carrier type, they can be reused to compose new coalgebras. To generate well-typed expressions as in Figure 5.2, taking `EqualF` as an example, the new coalgebra would be:

```
gEqualF' :: (MonadMaybe m, MonadRand m) => CoAlgM m EqualF (Type, Int)
gEqualF' (t, n) | t == TInt = none
                | n <= 0    = none
```

```
  | otherwise = do t <- elements [TInt, TBool]
                   return $ Equal (t, n - 1) (t, n - 1)
```

Such a generator can be composed with other well-typed generators by either $|*|_w$ or $|*|_d$. It is observed that by untangling production strategies with composition strategies, each part becomes reusable components, and can flexibly be combined in various ways.

Monad Transformers for Full Extensibility The use of monad transformers not only achieves reusability in coalgebras, but also provides a third dimension of extensibility: the functionality (computational behaviors) of programs is made extensible. In this way, more specialized combinators can be developed to encode new composition strategies.

5.1.3 An Overview of *SCCL*

Our methodology, to be presented in the following sections, shows that composition strategies should have the foundation in *natural transformations* from the product of functors to the sums-of-products. Moreover, specialized combinators can be derived from the natural transformations, and are directly applicable to coalgebras. The previous example on random generators has shown that, specialized combinators are responsible for encapsulating composition/distribution strategies without revealing the implementation to the users. As a consequence, client code can be simplified significantly.

Challenges of Modularization Although modularizing producers with coalgebras and specialized combinators offers many benefits, such modularization can be challenging in terms of implementation. Firstly, for different applications, the challenges can be different, and various composition strategies have to be identified and generalized into algorithms, patterns or modules. For example, for the random distributions, $|*|_w$ and $|*|_d$ have to deal with the weights, cardinalities, arities of functors, as well as the interaction with inputs. Also recall that in Chapter 4, we have identified three algorithmic challenges in modularizing parsing: left-recursion elimination, ordering and backtracking. A similar ordering issue also appears in modularizing *small-step semantic evaluation*, another application of producers. To avoid verbose manual ordering, *SCCL* encapsulates the strategy of a priority-based deterministic composition. Secondly, to ensure type-safety introduces additional difficulty in modularization. In *SCCL*, since functors are composed by binary co-products, the combinators are also designed in a binary form, which increases the difficulty of implementation.

Application of the *SCCL* Library Fortunately, *SCCL* has been designed for utility in the following three applications:

- Modular random generation;
- Modular small-step semantics;
- Modular monadic parsing.

A detailed description of *SCCL* is shown in Table 5.1. There are two basic strategies of composition. The list monad models non-determinism, and `|++|` merges all results together; this can be used in data enumerators. The `Maybe` monad models possible failure of computations, and correspondingly `|<|` is the combinator for selecting the first successful result. For the three applications,

Table 5.1: An overview of *SCCL*.

Modules		Monad Transformers	Monad	Combinators	Strategy
Basic Combinators		MonadMaybe	[]	|++|	Merging all possible results.
			Maybe	|<|	Selecting the first successful result.
Applications	Random Generation	RandT - MonadRand WeightT - MonadWeight	Uniform	$\|^*\|_u$	Random selection with uniform distribution.
			Weighted	$\|^*\|_w$, resetW	Weighted distribution on successful generators.
			Dynamic	$\|^*\|_d$, resetD	Dynamic distribution with size bound.
	Small-Step Semantics	PriorT - MonadPrior	Eval	|<>|, check	Single-step deterministic reduction.
	Monadic Parsing	ParserT - MonadParser	Parser	|<<|	First successful parser.

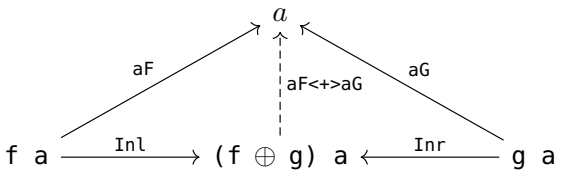

Figure 5.5: Generic combinator for algebras.

we define new monad transformers by using Haskell monad transformer library (MTL), construct specialized monads and design the corresponding combinators to meet our needs.

For quick reference, the type signatures of the combinators are presented in Figure 5.7. The first 7 are specialized coalgebra combinators, while the last 3 combinators provide auxiliary use in producers. Detailed definitions will be introduced later.

5.2 Composability of Coalgebras, and Product Forests

Our exploration begins from this section on the essence of modularizing *f-coalgebras*. The inspiration comes from the fact that Data Types à la Carte [Swierstra, 2008] implies a generic combinator for composing *f-algebras*. However, `f`-coalgebras are no longer naturally composable to form a co-product coalgebra, but instead they form a product coalgebra. Executing such a coalgebra results in a special form of data structure called a *product forest*. Section 5.3 then discusses how to extract the desired sums-of-product tree from a product forest.

5.2.1 The General Combinator for Coalgebras

Recall that Section 2.3.3 witnesses how DTC [Swierstra, 2008] achieves modular evaluation. Specifically, the evaluation algebra `evalAlgebra` is put under the type class `Eval`. By implementing an instance of `Eval` over the co-product type, it allows algebras to be automatically composed. An alternative for this is to make the composition explicit:

```
(<+>) :: (f a -> a) -> (g a -> a) -> (f ⊕ g) a -> a
(<+>) aF _  (Inl x) = aF x
```

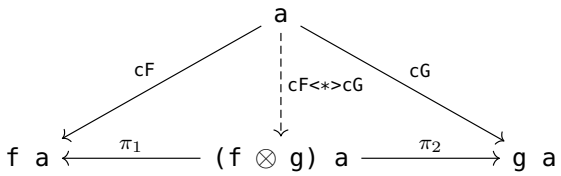

Figure 5.6: Generic combinator for coalgebras.

```
|++| :: CoAlgM [] f a -> CoAlgM [] g a -> CoAlgM [] (f ⊕ g) a

|<|  :: CoAlgM Maybe f a -> CoAlgM Maybe g a -> CoAlgM Maybe (f ⊕ g) a

|*|u :: (Cardinality f, Cardinality g) =>
        CoAlgM Uniform f a -> CoAlgM Uniform g a -> CoAlgM Uniform (f ⊕ g) a

|*|w :: Cardinality g =>
        CoAlgM Weighted f a -> CoAlgM Weighted g a -> CoAlgM Weighted (f ⊕ g) a

|*|d :: (Cardinality g, Arity f, Arity g, Derive a Int) =>
        CoAlgM Dynamic f a -> CoAlgM Dynamic g a -> CoAlgM Dynamic (f ⊕ g) a

|<>| :: CoAlgP h Eval f a -> CoAlgP h Eval g a -> CoAlgP h Eval (f ⊕ g) a

|<<| :: CoAlgM Parser f a -> CoAlgM Parser g a -> CoAlgM Parser (f ⊕ g) a

resetW :: Weighted a -> Weighted a
resetD :: Dynamic a -> Dynamic a
check  :: CoAlgP f Eval f (Fix f) -> CoAlgP f Eval f (Fix f)
```

Figure 5.7: *SCCL* combinators and their type signatures.

```
(<+>) _  aG (Inr x) = aG x
```

This generic combinator `<+>` combines an `f`-algebra and a `g`-algebra into an `(f ⊕ g)`-algebra. The resulting algebra can then be fed to the generic `fold` function, to perform a traversal on `Fix (f ⊕ g)` trees. This combinator is based on the type isomorphism between `(f a -> a) × (g a -> a)` and `(f ⊕ g) a -> a`, demonstrated by Figure 5.5 with a categorical-style diagram. Using such a combinator exposes the underlying mechanism of composition explicitly.

From the perspective of duality, it is natural to ask the question: can we build trees of type `Fix (f ⊕ g)`, given an `f`-coalgebra and a `g`-coalgebra?

Generally speaking, to produce such trees from the generic `unfold`, we need a coalgebra of type `a -> (f ⊕ g) a`. Unfortunately, `(a -> f a) × (a -> g a)` is not isomorphic to `a -> (f ⊕ g) a`, but to `a -> (f ⊗ g) a`. Here ⊗ is the *product* of two functors:

```
data (f ⊗ g) a = Prod (f a) (g a)
```

And the general combinator for coalgebras should be defined by:

```
(<*>) :: (a -> f a) -> (a -> g a) -> a -> (f ⊗ g) a
cF <*> cG = \a -> Prod (cF a) (cG a)
```

However, it is seemingly of little use. In practice, a composite functor usually has a "sum-of-products" form, as below:

$$f = a_1 \times a_2 \times ... \times a_n + b_1 \times b_2 \times ... \times b_m + ...$$

where each product represents a *constructor*. For example, the expression functor in Section 2.3.3 is denoted by `F(X) = Int + X * X`, combining the literal case with the addition case. The binary sum describes the extensibility of the abstract syntax, and hence each product is considered as an indivisible unit. When we define a coalgebra for generating expressions of height `n`, it would return

```
Add (n - 1) (n - 1)
```

to generate a complete binary tree, or return

```
Add 0 (n - 1)
```

for a linear tree. It does not make much sense to define coalgebras for the left and right sub-expressions separately. But if we really define coalgebras for every product, and compose them with `(<*>)`, the `unfold` will generate a "product-of-products" structure. What does such a structure look like?

5.2.2 Product Forests

To be more concrete, we define two functors for literals and the addition, separately. Also, we will use the type synonym below for coalgebras throughout this chapter.

```
type CoAlg f a = a -> f a

data LitF a = Lit Int
data AddF a = Add a a
```

We anticipate to generate a complete binary tree from input `(n, h)`, such that it has height `h` and denotes an expression whose sum is `n`. For that goal, two coalgebras are defined separately for `LitF` and `AddF` as follows:

```
hLit :: CoAlg LitF (Int, Int)
hLit (n, h) = Lit n

hAdd :: CoAlg AddF (Int, Int)
hAdd (n, h) = Add (n `div` 2, h - 1) (n `div` 2 + n `mod` 2, h - 1)
```

where `hLit` simply returns a literal of `n`, and `hAdd` tries to separate `n` in half for the two children of `Add`, with height `h` decreased by 1.

Now we intend to test the `(<*>)` combinator:

```
forest :: Fix (LitF ⊗ AddF)
forest =  unfold (hLit <*> hAdd) (11, 1)
```

And `forest` will have the following structure (code for pretty-printing omitted):

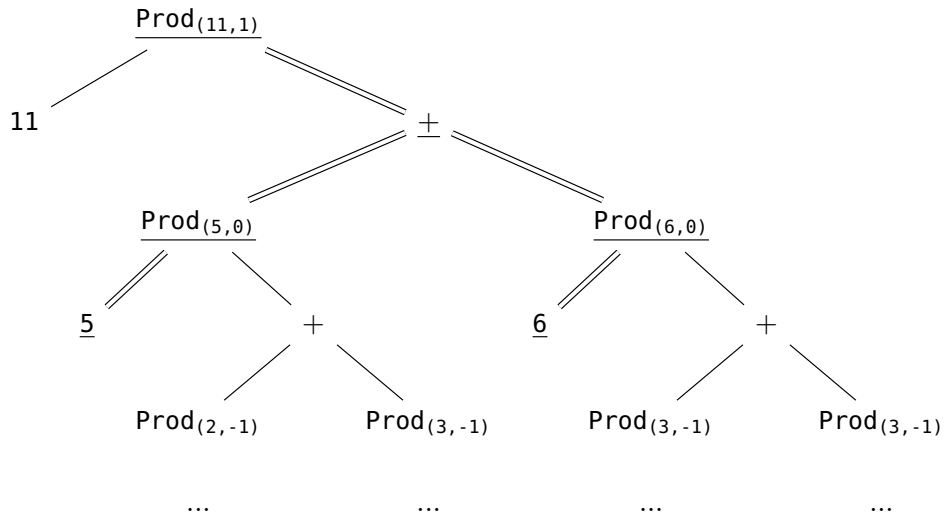

Figure 5.8: The structure of `forest`, generated by `unfold (hLit <*> hAdd) (11, 1)`. The subtree with underlined nodes and double-line edges reflects the expression `5 + 6`. And the subscript of `Prod` represents the input to the coalgebra at every node.

```
In (Prod (Lit 11)
         (Add (In (Prod (Lit 5) (Add (In (Prod (Lit 2) ...))
                                     (In (Prod (Lit 3) ...))))))
              (In (Prod (Lit 6) (Add (In (Prod (Lit 3) ...))
                                     (In (Prod (Lit 3) ...)))))))
```

It is, in fact, an infinite structure. The root of the tree is a pair consisting of `Lit 11`, generated by `hLit`, and a tree starting with `Add`, generated by `hAdd`. The latter tree has two subtrees, both of them also start with a pair. The left components of the pairs are respectively `Lit 5` and `Lit 6`, which are what the result would be if we choose to use `hLit` at this level. The right components are trees built using `Add`, as the results of `hAdd`.

Product Forests We call the structures of form `Fix (f ⊗ g ⊗ ...)` *product forests*, where the top-level functor has a product form. It is called a forest, rather than a tree, because it actually represents *a family of trees with sharing*.

It is hard to deal with this composite structure directly, but one may walk through the forest to obtain a more familiar sum-of-products tree, of type `Fix (LitF ⊕ AddF)`. In Figure 5.8, the subtree with underlined nodes and double-line edges stands for the tree `5 + 6`. It seems that we are making choices at every node of `Prod`. So what is the general method for extracting sums-of-product trees from a product forest?

5.3 From Product Forests to Sum-Of-Products

After `unfold` generates a product forest from the combined coalgebra, it is necessary to walk through the forest, make a decision at every node of `Prod`, and consequently recover a desired sum-of products

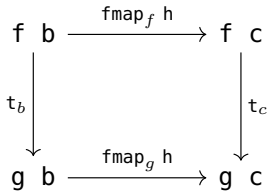

Figure 5.9: Naturality of a transformation.

tree, which truly represents the composite abstract syntax. Such a decision maker can be realized by a *natural transformation*.

5.3.1 Natural Transformation

A *natural transformation* `t :: f ⇢ g` is a collection of arrows that, for each type `a`, transforms a structure `f a` to `g a` in a way such that the following naturality holds for any function `h :: b -> c`:

$$\mathtt{t}_c \,.\, \mathtt{fmap}_f\ \mathtt{h} \;=\; \mathtt{fmap}_g\ \mathtt{h} \,.\, \mathtt{t}_b$$

It is also depicted in Figure 5.9 with a categorical diagram. Operationally, a natural transformation can be understood as a transformation that changes only the shape, but not the content of the structure. In Haskell, this is modelled by the following type synonym:

```
type f ⇢ g = forall a. f a -> g a
```

where the universal quantification ensures that a program having this type will not examine the elements of type `a` stored in `f`.

Extracting Trees from Product Forests A natural transformation `t :: f ⇢ g` can be used to convert `Fix f` to `Fix g` by either a fold or an unfold:

Theorem 1. *Let `t :: f ⇢ g`, we have that:*

```
fold (In . t) = unfold (t . out)
```

Proof. Immediate from the *hylo-shift law* [Hinze et al., 2011]. □

As a corollary, a `Fix (f ⊗ g)` value can be converted to a `Fix (f ⊕ g)` value by either a fold or an unfold, if we have a natural transformation from the product to the co-product:

Corollary 2. *Let `cF :: CoAlg f a`, `cG :: CoAlg g a`, and `t :: f ⊗ g ⇢ f ⊕ g`. We have*

```
fold (In . t) . unfold (cF <*> cG) = unfold (t . out) . unfold (cF <*> cG)
```

Unfold-Transformation Pattern The above corollary suggests a pattern for extracting a sum-of-products tree from a product forest, in the following steps:

- Composing coalgebras with the generic combinator `<*>`;
- Feeding the composed coalgebra to `unfold` to produce a product forest;
- Applying a fold/unfold-based transformation, to get the sum tree.

5.3.2 Deforesting Product Forests

Unfortunately, the unfold-transformation pattern has two main drawbacks:

I. It produces intermediate product forests. In the general case, modularizing coalgebras requires building product forests. However, these product forests only appear in the middle of computation, but are absent in the final result. It would be good to eliminate/deforest the product forest, which may benefit us in terms of time and space efficiency.

*II. It is hard to transform product forests with **n**-ary products.* In the transformation/selection function there are two functors involved:

```
t :: f ⊗ g ↛ f ⊕ g
```

The function is generic on `f` and `g`. Nevertheless, if a product forest is built on the nested composition of three coalgebras,

```
triForest :: Fix (f1 ⊗ (f2 ⊗ f3))
```

one layer of transformation can only transform the outermost product:

```
unfold (t . h) triForest :: Fix (f1 ⊕ (f2 ⊗ f3))
```

For a deep transformation, one option is to tediously define transformation functions for different arities. For example:

```
t3 :: f ⊗ (g ⊗ h) ↛ f ⊕ (g ⊕ h)
```

Although `t3` can be generated by `t`, it is still inconvenient to use since the arity can be unfixed. A better approach could be using the list-of-functor approach by [Oliveira et al., 2015] to replace the binary products and co-products, so that we obtain a global view of all functors. Nevertheless, it still requires cumbersome code. While this drawback is more of a technical matter, it requires significant additional sophistication to be dealt with properly.

Both of the drawbacks motivate us to simplify the construction of sum-of-products by deforestation [Wadler, 1988]. Formally,

Theorem 3. *For all* `h :: CoAlg f a` *and* `t :: f ↛ g`*, we have that*

```
unfold (t . out) . unfold h = unfold (t . h)
```

Proof. This is called *functor fusion law* for unfolds in [Hinze, 2010]. A simple proof uses the *ana-fusion law* from [Hinze, 2010; Pardo, 1998]:

```
h . k = fmap k . h'  ⟹  unfold h . k = unfold h'
```

It suffices to prove that `t . out . unfold h = fmap (unfold h) . t . h`:

```
   t . out . unfold h
=    {- definition of unfold h -}
   t . fmap (unfold h) . h
=    {- naturality of t -}
   fmap (unfold h) t . h
```

□

Corollary 4. *Let* `cF :: CoAlg f a`, `cG :: CoAlg g a`, *and* `t :: f ⊗ g ⇢ f ⊕ g`. *We have*

```
fold (In . t) . unfold (cF <*> cG) = unfold (t . out) . unfold (cF <*> cG)
                                   = unfold (t . (cF <*> cG))
```

The above corollary shows that the unfold-transformation pattern can be replaced with a single unfold, where the natural transformation is earlier applied to the product coalgebra. It simplifies and enhances the efficiency of construction.

5.3.3 Discussion

A natural transformation plays the role of a selector, which reflects some certain *selection strategy* in real applications. Nevertheless, it is worth noticing that, since naturality is the pre-condition of deforestation, our selection strategies are restricted to the following scenarios:

I. The selection is determined by functors or constructors. There are not many examples for transformations of type `t :: f ⊗ g ⇢ f ⊕ g`. One could be the following function that always chooses the first functor in the product:

```
alwaysFirst :: f ⊗ g ⇢ f ⊕ g
alwaysFirst (Prod x _) = Inl x
```

which is not very useful in practice. While `alwaysFirst` is polymorphic on `f` and `g`, a specialized selector can instantiate both functors, and perform pattern-matching on their constructors respectively. It is, however, considered ad-hoc and harmful, thus is not common in practice.

II. The selection is determined by inputs. Although inputs are not involved in the previous theorems, we can generalize the definition of product to store input values:

```
data (f ⊗ g) a b = Prod a (f b) (g b)
```

while the new generic combinator becomes

```
(<*>) :: CoAlg f a -> CoAlg g a -> CoAlg ((f ⊗ g) a) a
cF <*> cG = \a -> Prod a (cF a) (cG a)
```

Correspondingly, Theorem 3 can be adapted to the transformation

```
t :: (f ⊗ g) a ⇢ f ⊕ g
```

as well, and the difference is that now `t` also allows inputs to make the decision.

III. The selection involves partiality of functions. One way to select from several coalgebras is to think of them as partial functions: some of the coalgebras may fail, and we wish to choose the first one that succeeds. In Haskell, partiality is often modelled by `Maybe` so as to catch the failure. The first step is to introduce the composition of functors:

```
data (f • g) a = Comp { unComp :: f (g a) }

instance (Functor f, Functor g) => Functor (f • g) where
  fmap h = Comp . fmap (fmap h) . unComp
```

Then a `f`-coalgebra that may fail can be represented by a `(Maybe • f)`-coalgebra. For instance, variants of `hLit` and `hAdd` in Section 5.2.2 are defined as below:

```
hLit' :: CoAlg (Maybe • LitF) (Int, Int)
hLit' (n, h) | h <= 0    = Comp . Just $ Lit n
             | otherwise = Comp Nothing

hAdd' :: CoAlg (Maybe • AddF) (Int, Int)
hAdd' (n, h) = Comp . Just $ Add (n `div` 2, h - 1)
                                 (n `div` 2 + n `mod` 2, h - 1)
```

where `hLit'` rejects any positive `h` compared with `hLit`. And the following natural transformation represents our "first-succeed" strategy:

```
firstSucceed :: (Maybe • f) ⊗ (Maybe • g) ↛ f ⊕ g
firstSucceed (Prod (Comp (Just x)) _              ) = Inl x
firstSucceed (Prod _               (Comp (Just y))) = Inr y
```

which is possible to raise runtime exceptions on non-exhaustiveness. As a result, by Theorem 1 and Theorem 3,

```
    fold   (In . firstSucceed)  . unfold (hLit' <*> hAdd')
=   unfold (firstSucceed . out) . unfold (hLit' <*> hAdd')
=   unfold (firstSucceed . (hLit' <*> hAdd'))
```

Applying the above function to `(11, 1)` yields expression `(5 + 6)`, of type `Fix (LitF ⊕ AddF)`. In this way, our initial goal is achieved, namely to generate a complete binary tree of a certain height and certain sum.

Besides the above scenarios, in a more general case, it is possible to define more specialized transformations that have knowledge of the overall structure, together with all the content inside the product forest. That is to say, such transformations are non-natural. Fortunately, our study supports that natural transformations are more commonly used in practice, but have to involve side effects that take the main responsibility for selection. The above partiality example is actually a special case of side effects, but it succeeds in representing a general selection strategy. It is used here to whet the appetite before we introduce more exploration on monadic variants in later sections, together with some specialized combinators in our library with applications.

5.4 Monadic Variants

In this section, we generalize the approach for composing coalgebras to the effectful world. While pure coalgebras are restricted in terms of practicality, with monadic coalgebras, the side effects take the responsibility for realizing sophisticated selection strategies, and deriving specialized combinators for certain operations with proper encapsulation of strategies.

```
type AlgM m f a = f a -> m a

foldM :: (Traversable f, Monad m) => AlgM m f a -> Fix f -> m a
foldM h = h <=< mapM (foldM h) . out
```

Figure 5.10: The generic monadic fold.

```
type CoAlgM m f a = a -> m (f a)

unfoldM :: (Traversable f, Monad m) => CoAlgM m f a -> a -> m (Fix f)
unfoldM h = fmap In . (mapM (unfoldM h) <=< h)
```

Figure 5.11: The generic monadic unfold.

5.4.1 Monadic Folds and Unfolds

MONADIC ALGEBRAS A monadic algebra has type signature `f a -> m a`, where `f` is a functor, `m` a monad, and `a` is the carrier. For instance, when `m` is the `Maybe` monad, an algebra can represent evaluation on arithmetic expressions with possible failure. Generalizing the generic fold, a *monadic catamorphism* requires a lifting of functor `f` in [Pardo, 1998], while in Haskell this is achieved by making `f` an instance of `Traversable`, so as to allow applications to be performed and collected over the functor. An implementation is given in Figure 5.10. Noticeable is the use of `mapM`, provided by `Traversable f`. The fish operator `<=<` denotes the right-to-left Kleisli composition.

It is worth mentioning that monadic algebras are still naturally composable as pure ones; the general combinator below witnesses one direction of the type isomorphism:

```
(<++>) :: AlgM m f a -> AlgM m g a -> AlgM m (f ⊕ g) a
(<++>) aF _  (Inl x) = aF x
(<++>) _  aG (Inr x) = aG x
```

MONADIC COALGEBRAS A monadic coalgebra has type `a -> m (f a)`, given monad `m`, functor `f` and carrier `a`. [Pardo, 1998] proposed *monadic anamorphism* as the dual operation to monadic catamorphism. In the context of Haskell, an implementation is presented in Figure 5.11. Just like `unfold`, the coalgebra is applied first to the input, resulting in a top-down procedure.

Having known that the composition of pure coalgebras results in a product type, we need to figure out how such composability is generalized to the monadic case.

5.4.2 General Combinator for Monadic Coalgebras

For the composition of two monadic coalgebras, one might consider to have a combinator with the following signature:

```
comp :: CoAlgM m f a -> CoAlgM m g a -> CoAlgM m (f ⊗ g) a
```

But somewhere it requires to produce `m ((f ⊗ g) a)` from `m (f a)` and `m (g a)`. One possible implementation could be:

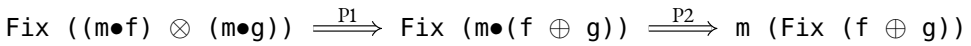

Figure 5.12: Two-step transformation from a monadic product forest to a monadic sum tree.

```
comp :: Applicative m => CoAlgM m f a -> CoAlgM m g a -> CoAlgM m (f ⊗ g) a
comp f g = \a -> liftA2 Prod (f a) (g a)
```

which sequentially composes two monadic values as applications. Nevertheless, this does not represent a general composition; it mixes the effects of both values in an irreversible way and often behaves unexpectedly. For example, when `m` is the list monad, a lot of duplicate results are produced. The real solution is to keep the two effects independent with a pair, and delegate effect handling to a subsequent transformation. It is due to the isomorphism between `(a -> m (f a)) × (a -> m (g a))` and `a -> (m (f a), m (g a))`.

Instead of defining a new datatype for the pair, we can reuse the product combinator (⊗) with functor composition (•), to model monadic coalgebras with pure ones. The general combinator for pure coalgebras can be reused here:

```
(<*>) :: CoAlg (m•f) a -> CoAlg (m•g) a -> CoAlg ((m•f) ⊗ (m•g)) a
cF <*> cG = \a -> Prod (cF a) (cG a)
```

This is connected to the previous study and is thus helpful for reusing the properties.

5.4.3 Flow of Construction and Deforestation

Given monadic coalgebras of functor `f` and functor `g`, our final desired result would be a sum-of-products tree with effects, namely with type `m (Fix (f ⊕ g))`. Yet the above general combinator generates a `Fix ((m•f) ⊗ (m•g))` value from `unfold`. This is an *effectful product forest*, with side effects hidden inside. As before, we expect a transformation to select the sum tree from product forest. Seemingly, both `foldM` and `unfoldM` are eligible:

```
    foldM alg :: Fix ((m•f) ⊗ (m•g)) -> m (Fix (f ⊕ g))
unfoldM coalg :: Fix ((m•f) ⊗ (m•g)) -> m (Fix (f ⊕ g))
```

To explore it further, we expand the single step into two steps (see Figure 5.12):

- P1: a pure transformation from the product forest, to the fixpoint of monadic co-product. This step does not thread side effects.
- P2: a traversal throughout the structure, to evaluate the monadic actions and collect the results. This step threads the effects and finally produces the desired monadic sum tree.

The more interesting part is P1, as it is where selection strategies are involved. P2 can be accomplished by a simple `unfoldM` operation to thread the effects throughout the structure. As a whole, the complex unfold-transformation pattern can be implemented by a single function in Figure 5.13. Note that the `unfoldM` relies on an instance of `Traversable` over `(f ⊕ g)`, which can be automatically derived by Haskell.

```
type Trafo m f g = (m•f) ⊗ (m•g) ⇾ m•(f ⊕ g)

build :: (Monad m, Traversable f, Traversable g) =>
         CoAlg (m•f) a -> CoAlg (m•g) a -> Trafo m f g -> a -> m (Fix (f ⊕ g))
build cF cG t = unfoldM (unComp . out) . unfold (t . out) . unfold (cF <*> cG)
```

Figure 5.13: The implementation of monadic unfold-transformation pattern.

DEFORESTATION The previous deforestation technique in Section 5.3.2 is also adaptable here to replace the unfold-transformation pattern. The formal theorem is presented as follows:

Theorem 5. *For all* `h :: CoAlg f a` *and* `t :: forall x. f x -> m (g x)`*, we have*

$$\texttt{unfoldM (t . out) . unfold h} = \texttt{unfoldM (t . h)}$$

Proof. This is a generalization of Theorem 3 for monadic anamorphisms. To prove it we use the *monadic ana-fusion law* in [Pardo, 1998], which states that for all `j :: b -> m (g b)`, `k :: a -> b`, `j' :: a -> m (g a)`, we have

$$\texttt{j . k} = \texttt{fmap}_m\ (\texttt{fmap}_g\ \texttt{k})\ \texttt{. j'} \implies \texttt{unfoldM j . k} = \texttt{unfoldM j'}$$

For this case we need `t . out . unfold h` $=$ `fmap`$_m$ `(fmap`$_g$ `(unfold h)) . t . h`. The proof goes:

```
    t . out . unfold h
=     {- definition of unfold -}
    t . fmap_f (unfold h) . h
=     {- naturality of t -}
    fmap_m (fmap_g (unfold h)) . t . h
```

□

Corollary 6. *Given two monadic coalgebras and a natural transformation* `t :: Trafo m f g`*, the* `build` *function in Figure 5.13 can be deforested into a single unfold, since*

```
    build cF cG t
=     {- definition of build -}
    unfoldM (unComp . out) . unfold (t . out) . unfold (cF <*> cG)
=     {- Theorem 3 -}
    unfoldM (unComp . out) . unfold (t . (cF <*> cG))
=     {- Theorem 5 -}
    unfoldM (unComp . t . (cF <*> cG))
```

```
enumLit :: CoAlg ([] • LitF) ([Int], Int)
enumLit (xs, n) = Comp $ map Lit xs

enumAdd :: CoAlg ([] • AddF) ([Int], Int)
enumAdd (xs, n) | n <= 0    = Comp []
                | otherwise = Comp [Add (xs, n - 1) (xs, n - 1)]
```

Figure 5.14: Enumerators of `LitF` and `AddF` up to a certain depth.

Example When `m` is instantiated to the list monad, the operations are able to describe a collection of results, which exposes the underlying *non-determinism*. Following the previous arithmetic expression language in Section 5.2.2, an example of monadic coalgebras is defining enumerators that exhaustively enumerate all valid expressions up to a certain depth. Figure 5.14 defines two coalgebras of `LitF` and `AddF` respectively. Note that an input `(xs, n)` implies that the generated expressions have depth at most `n`, and their leaves (literals) are extracted from list `xs`.

Since we intend to collect all valid results, the selection strategy here is to combine the results from both sides by concatenation. The natural transformation is as follows:

```
mergeResults :: Trafo [] f g
mergeResults (Prod (Comp fs) (Comp gs)) = Comp $ map Inl fs ++ map Inr gs
```

Consequently the final enumerator of the combined language is obtained either from the unfold-transformation pattern, or the deforestation approach:

```
genEnum :: ([Int], Int) -> [Fix (LitF ⊕ AddF)]
genEnum = build enumLit enumAdd mergeResults

genEnum' :: ([Int], Int) -> [Fix (LitF ⊕ AddF)]
genEnum' = unfoldM (unComp . mergeResults . (enumLit <*> enumAdd))
```

Feeding `([1, 2], 1)` to either function yields `[1, 2, (1 + 1), (1 + 2), (2 + 1), (2 + 2)]`, a list of all 6 expressions within depth 1 as expected.

5.4.4 Discussion

This section gives more insights into monadic unfolds, and draws a connection to the approach in Chapter 4.

How About `foldM`? We have not mentioned the fold-based transformation in monadic unfolds. Looking back at Figure 5.12, P1 can instead be a fold-transformation, since Theorem 1 ensures the equivalence. However, P2 cannot be an instance of `foldM`. On the one hand, `foldM` requires the constraint `Traversable (m•(f ⊕ g))`, but it is not clear how it holds for an arbitrary `m`. On the other hand, even if `foldM` is applicable, Theorem 1 does not immediately carry over to a monadic setting. Someone might assume that

```
foldM (fmap In . t) = unfoldM (t . out)
```

for any natural transformation `t`, this does not hold in general. In the pure setting, the naturality is the key property to ensure Theorem 1. Whereas `t :: forall x. f x -> m (g x)` is not sufficient to guarantee the corresponding property

$$\texttt{t}_{Fixg}\ \texttt{=<<}\ \texttt{mapM}_f\ \texttt{h z}\ =\ \texttt{mapM}_g\ \texttt{h}\ \texttt{=<<}\ \texttt{t}_{Fixf}\ \texttt{z}$$

where `z :: f (Fix f)`, and `h :: Fix f -> m (Fix g)` standing for the recursive transformation. The real reason for this inequality comes from the essence of `foldM` and `unfoldM`: one threads and collects effects in a bottom-up manner, while the other does the same top-down. Also in the above property, we could see that the left-hand side (for `foldM`) applies traversal and sequencing before the transformation, while the right side (for `unfoldM`) goes in the opposite direction.

EXAMPLE A quick counter-example for the equality is given below. For the transformation, we instantiate both `f` and `g` to `LitF ⊕ AddF`, and `m` to the list monad. In such a self-transformation `dup` duplicates each branch of `Add` and returns two additions in total, but puts any `Lit` into a singleton list:

```
dup :: (LitF ⊕ AddF) a -> [(LitF ⊕ AddF) a]
dup n@(Inl _)       = [n]
dup (Inr (Add x y)) = [Inr $ Add x x, Inr $ Add y y]
```

Now feeding `"1 + (2 + 3)"` to `"foldM (fmap In . dup)"` yields

```
[1+1, (2+2)+(2+2), 1+1, (3+3)+(3+3)]
```

Yet the same input on `"unfoldM (dup . out)"` gives

```
[1+1, (2+2)+(2+2), (2+2)+(3+3), (3+3)+(2+2), (3+3)+(3+3)]
```

Step-by-step evaluation is presented in Figure 5.15.

TWO-STEP TRANSFORMATION? Figure 5.12 uses a two-step transformation to describe the process of selecting an effectful sum tree from a product forest, since they play different roles therein. The second step, however, is merely an identity traversal over the structure to collect the effects. The reason for separating this step is that, in some special situations, we may not anticipate to perform such a traversal.

Recall the enumeration example in Section 5.4.3. For the list monad, P2 behaves like a flattening; while `Fix ([]•f)` stores shared structures, the `unfoldM` expands the effectful structure into a list of expressions. It is equivalent to the following recursive function:

```
flatten :: Traversable f => Fix (m•f) -> m (Fix f)
flatten (In (Comp xs)) = [In y | x <- xs, y <- mapM flatten x]
```

which uses *monad comprehensions* [Wadler, 1990]. Now suppose that we implement an enumerator for a real-world language, which incorporates a large number of functors (constructors). The enumerator will generate a lot of tree structures, but there is considerable sharing on sub-trees. In that case, it would be much inefficient to check properties on the large list produced by `flatten`, because shared structures will be copied during flattening, consequently the calculations of property checking will be duplicated. Instead, for those calculations that can be modelled by bottom-up traversals, one may implement a `f`-algebra, lift it into an effectful `([]•f)`-algebra, and directly apply it to `Fix ([]•f)` with `fold`. That results in a considerable reduction to the execution time.

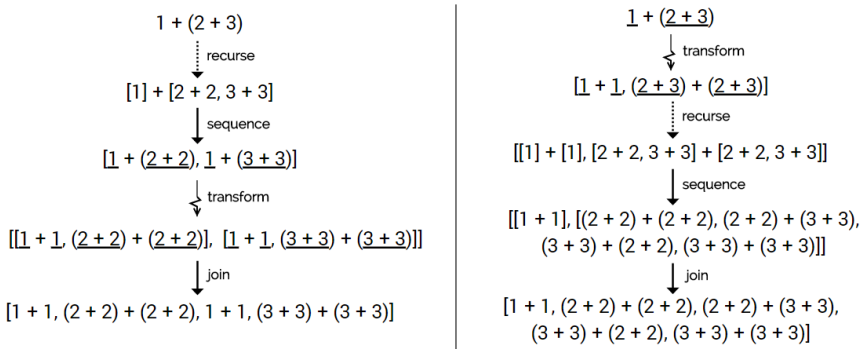

Figure 5.15: Evaluation steps of `foldM`-based transformation (left) and `unfoldM`-based transformation (right), with input `1 + (2 + 3)`. The underlines highlight what `dup` does.

Comparison with Type-Safe Modular Parsing In Chapter 4 we have introduced our approach for modular parsing in Scala. In fact, parsing is only one representative among the applications of building structures (producers). While this chapter discusses a more general composition of coalgebras in functional programming, it is interesting to observe the essence and connection between both approaches across different programming languages.

In short, the Scala approach uses a more general recursion pattern to modularize producers, while the Haskell approach aims at modularizing coalgebras with generic unfolds. Recall that in the Scala approach, an extensible parser has the following type signature:

```
def parser[E]: Alg[E] => Parser[E]
```

where `Alg[E]` represents an abstract syntax tree from an Object Algebra interface, and `Parser[E]` stands for a packrat parser. The type parameter `E` makes `parser` polymorphic. This function does not require a **`String`** parameter for the input, for it has been incorporated into `Parser`. To put it more general, an extensible producer has the following type:

```
def build[E]: A => Alg[E] => E
```

where `A` is the input type. In Object Algebras, an object is represented by a generic function `Alg[E] => E` rather than a real structure. Although we could connect this type signature to *hylomorphisms* in functional programming, it is better to understand that `Alg[E] => E` is isomorphic to `Fix f` in Haskell. Furthermore, the extensibility of `parser` also relies on inheritance and dynamic dispatch in Scala, for which open recursion is the alternative in Haskell. To summarize, an equivalent form in Haskell would be:

```
type Open x = x -> x

build :: Open (a -> Fix f)
```

and a combinator would have type

```
compose :: Open (a -> Fix f) -> Open (a -> Fix f) -> Open (a -> Fix f)
```

While modularizing coalgebras requires a top-down unfold, `compose` is able to perform bottom-up selection strategies, since `Fix f` structures are accessible. This corresponds to a more specialized (non-natural) transformation in the unfold-transformation pattern.

Although `compose` is eligible for more general situations, it has to sacrifice some properties due to such generality. In contrast, recursion schemes like `unfold` separate the generic recursion from the definition of coalgebras, thus they guarantee more properties including productivity, since a coalgebra `a -> f a` always produces its outermost constructor without any consumption. Consequently, programs get easier to be reasoned about and optimized.

ENCAPSULATION The Scala code encapsulates parsing into the `Parser` type, and defines combinators over that type. Similarly, we can port the enumeration example in Section 5.4.3 to Scala as follows:

```
trait Enum[E] {
  def enum(i: Int): List[E]
}

def or[E](x: Enum[E], y: Enum[E]) = new Enum[E] {
  def enum(i: Int) = x.enum(i) ++ y.enum(i)
}
```

Correspondingly, an enumerator has type `Alg[E] => Enum[E]`, and is extensible via the `or` combinator.

5.5 Implementation of *SCCL*

This section presents the implementation of our *SCCL* library. It starts with two basic combinators, known to express some fundamental strategies of selection to be used in more complicated behaviors. Then the basic infrastructure adopts *monad transformers* [Liang et al., 1995] to make the functionality extensible in effectful computations, and also to make coalgebras reusable. Besides the use of Haskell monad transformer library (MTL), *SCCL* further combines some transformers and packs them into new monad transformers for general purpose, and later they are used for our applications: *random generation*, *small-step evaluation*, and *monadic parsing*. *SCCL* defines several specialized combinators for those applications, all of which are based on effectful natural transformations. The design of *SCCL* aims to convey the idea that, for those unfold-shape operations that build data structures, users have the power to achieve:

- extensibility on functionality;
- reusability of coalgebras;
- modularization in selection strategies.

Furthermore, specialized combinators can be designed to free users from explicitly handling composition in a cumbersome way.

5.5.1 Basic Combinators

The two basic combinators are specialized for `[]` and `Maybe` respectively.

I. LIST MONAD: MERGING ALL POSSIBLE RESULTS. In Section 5.4.3, `mergeResults` is the transformation that carries out such a strategy. While the `Applicative` instance of list expands nondeterminism in the form of a Cartesian product, it performs concatenation here just as the `Alternative` or `MonadPlus` instance does.

Deriving Specialized Combinators By Corollary 6, we can use the general coalgebra combinator `|*|` for composition, followed by `mergeResults` to produce a monadic sum coalgebra directly, before it is fed to `unfoldM`. Fusing the general combinator and the transformation gives us a second deforestation: the product type can be eliminated. In general, a natural transformation `t` results in a combinator that directly composes two monadic coalgebras `CoAlgM m f a` and `CoAlgM m g a` into `CoAlgM m (f ⊕ g) a`:

```
(|*|_t) :: CoAlgM m f a -> CoAlgM m g a -> CoAlgM m (f ⊕ g) a
```

And one possible definition of this combinator, building directly on `t :: Trafo m f g`, is:

```
cF |*|_t cG = \x -> unComp . t $ Prod (Comp $ cF x) (Comp $ cG x)
```

Taking `mergeResults` as an example, the specialized combinator `|++|` would be:

```
   cF |++| cG
=     {- derivation from mergeResults -}
   \x -> unComp . mergeResults $ Prod (Comp $ cF x) (Comp $ cG x)
=     {- expanding mergeResults: fs := cF x, gs := cG x -}
   \x -> map Inl (cF x) ++ map Inr (cG x)
```

Such a combinator can be used to implement data enumerators.

II. **`Maybe`** MONAD: SELECTING THE FIRST SUCCESSFUL RESULT. When both coalgebras give optional values, one strategy is to choose the first successful result in order:

```
(|<|) :: CoAlgM Maybe f a -> CoAlgM Maybe g a -> CoAlgM Maybe (f ⊕ g) a
cF |<| cG = \x -> fmap Inl (cF x) `mplusMaybe` fmap Inr (cG x)
  where Just x  `mplusMaybe` _ = Just x
        Nothing `mplusMaybe` y = y
```

The `mplusMaybe` behaves the same as `mplus` of the `Maybe` monad. At this point, one may feel that a monadic combinator (or its natural transformation) can be generalized with the `MonadPlus` instance. The generalization could be as follows:

```
combine :: MonadPlus m => CoAlgM m f a -> CoAlgM m g a -> CoAlgM m (f ⊕ g) a
cF `combine` cG = \x -> fmap Inl (cF x) `mplus` fmap Inr (cG x)
```

Although it seems that both `|++|` and `|<|` can rely on `MonadPlus`, we argue that it does not work in general. For example, the `mplus` of `IO` is a function that deals with `IOError` exceptions. It is unclear how it can be used in real applications. Furthermore, in the following sections, some combinators will need to identify the functors for grasping necessary information; the functors are hidden in `mplus`, however. Under those circumstances, individual `mplus` implementations are considered unreliable.

```
class Monad m => MonadRand m where
  seed :: m StdGen
  choose :: Random a => (a, a) -> m a
  choose range = liftM (fst . randomR range) seed
  elements :: [a] -> m a
  elements [] = error "elements applied to empty list"
  elements xs = (xs !! ) `fmap` choose (0, length xs - 1)
  frequency :: [(Int, m a)] -> m a
  frequency [] = error "frequency applied to empty list"
  frequency xs = choose (1, total) >>= (`pick` xs)
    where total = sum (map fst xs)
          pick n ((k, x):ys) | n <= k    = x
                             | otherwise = pick (n - k) ys
  binomial :: (Double, m a) -> (Double, m a) -> m a
  binomial (w1, x) (w2, y) = do
    r <- choose (0, 1)
    if r < w1 / (w1 + w2) then x else y

instance Monad m => MonadRand (RandT m) where
  seed = RandT $ do s <- get
                    let (s1, s2) = split s
                    put s2
                    return s1
```

Figure 5.16: The `MonadRand` library for randomness.

5.5.2 Application I: Random Generation

The first application of modular unfolds is modularizing random data generators. For different features, *SCCL* adopts different strategies represented by combinators.

5.5.2.1 Uniform Distribution

Besides the basic combinators, randomness is in fact another fundamental and commonly-used selection strategy. *SCCL* defines `RandT` as the transformer for randomness, and `MonadRand` for the library of auxiliary functions.

The Random Monad Transformer The definition of `RandT` contains a `StdGen`[1] state for the random seed:

```
newtype RandT m a = RandT { unRandT :: StateT StdGen m a }
```

The instances for `Functor`, `Applicative`, `Monad` and `MonadTrans` are straightforwardly implemented; thus detailed code is omitted here.

[1] The random seed type comes from `System.Random` in Haskell.

```
class Cardinality (f :: * -> *) where
  card :: Proxy f -> Int

instance {-# OVERLAPPABLE #-} Cardinality (f ⊕ g) where
  card _ = card (Proxy :: Proxy f) + card (Proxy :: Proxy g)

instance {-# OVERLAPPABLE #-} Cardinality f where
  card _ = 1

getCard :: Cardinality f => CoAlgM m f a -> Int
getCard = card . proxy

proxy :: CoAlgM m f a -> Proxy f
proxy _ = Proxy
```

Figure 5.17: The code for figuring out the cardinality of a functor.

Then it comes the `MonadRand` interface, presented in Figure 5.16. Some of the names are borrowed from `MonadGen` and the QuickCheck [Claessen and Hughes, 2000] library. There is also some similarity to the `QuickCheck-GenT`[1] library. Compared with `GenT`, `RandT` removes the size parameter, for sizes are supposed to be maintained by coalgebras in the carrier type. Additionally, while `GenT` splits the random seed on every bind operation, `RandT` defines the `seed` function for querying random seeds explicitly, and the state is updated on each query so as to prevent unexpected identical copies. Moreover, noticeable is the `binomial` function that chooses between two monadic values given their weights. This function will be used later in our specialized combinators.

Cardinality We would like to define a binary combinator that composes an `f`-coalgebra with a `g`-coalgebra. But it is incorrect if we set equal weights to `f` and `g` for random selection, because there can be multiple functors, and hence in the nested composition (say right-associative), the probability distribution can be $[\frac{1}{2}, \frac{1}{4}, ...]$. To achieve a uniform random distribution globally over all functors, we need to figure out the *cardinality* of `f` and `g`, namely the number of individual functors included, and arrange their weights accordingly. Our approach calculates this automatically from the type information in Figure 5.17.

In Figure 5.17, `Cardinality` captures cardinality with a type class, and its instances pattern-match on the functor. With a co-product form `f ⊕ g`, the total cardinality is the cardinality of `f` added to that of `g`. By default, an individual functor has cardinality `1`. Finally, `getCard` is able to grasp the cardinality from a coalgebra of a certain functor `f`, while `proxy` identifies and delivers the functor type.

Strategy: Random Selection with a Uniform Distribution. When instantiating the monad to:

```
type Uniform = RandT Identity
```

[1] `http://hackage.haskell.org/package/QuickCheck-GenT`

The following monadic natural transformation realizes random selection by cardinality:

```
uniformTrafo :: (Cardinality f, Cardinality g) => Trafo Uniform f g
uniformTrafo (Prod (Comp fs) (Comp gs)) =
  Comp $ frequency [(card (getProxy fs), fmap Inl fs),
                    (card (getProxy gs), fmap Inr gs)]
  where getProxy :: m (h a) -> Proxy h
        getProxy _ = Proxy
```

where `frequency` applies a weighted random distribution. Since the weights are based on cardinality, this binary composition ensures a uniform distribution over functors. Consequently, the specialized combinator $|*|_u$ is derived by:

```
(|*|u) :: (Cardinality f, Cardinality g) =>
    CoAlgM Uniform f a -> CoAlgM Uniform g a -> CoAlgM Uniform (f ⊕ g) a

   cF |*|u cG
=     {- derivation from uniformTrafo -}
   \x -> unComp . uniformTrafo $ Prod (Comp $ cF x) (Comp $ cG x)
=     {- expanding uniformTrafo: fs := cF x, gs := cG x -}
   \x -> frequency [(card (getProxy (cF x)), fmap Inl (cF x)),
                    (card (getProxy (cG x)), fmap Inr (cG x))]
=     {- substituting with the equivalent getCard -}
   \x -> frequency [(getCard cF, fmap Inl $ cF a)
                    (getCard cG, fmap Inr $ cG a)]
```

Guaranteed by the naturality of `uniformTrafo`, $|*|_u$ is generic on the carrier type, and hence is offered reusability.

Run a Generator Finally, after coalgebras are composed by $|*|_u$ into a single coalgebra of the sum-of-products functor, the following function runs the generator by applying the generic unfold, obtaining the resulting `Uniform` value and evaluating it in `IO`:

```
generateU :: Traversable f => a -> CoAlgM Uniform f a -> IO (Fix f)
generateU input coalg = newStdGen >>= runUniform
  where runUniform g = return . runIdentity . flip evalStateT g .
                       unRandT $ unfoldM coalg input
```

5.5.2.2 Weighted Random Distribution with Failure

We continue to generalize the uniform distribution to a weighted random distribution. Moreover, generators are allowed to have failure.

The Weight Monad Transformer The following `WeightT` transformer defines the effects needed for a weighted distribution:

```
newtype WeightT m a = WeightT {
  unWeightT :: StateT (Int, Double) (ReaderT [Double] m) a
}
```

```
class Monad m => MonadWeight m where
  pointer   :: m Int
  succ      :: m ()
  reset     :: m ()
  getWeight :: Int -> m Double
  getAcc    :: m Double
  setAcc    :: Double -> m ()

instance Monad m => MonadWeight (WeightT m) where
  pointer     = WeightT $ get >>= return . fst
  succ        = WeightT $ modify $ \(p, acc) -> (p + 1, acc)
  reset       = WeightT $ put (0, 0)
  getWeight p = WeightT $ ask >>= return . (!! p)
  getAcc      = WeightT $ get >>= return . snd
  setAcc w    = WeightT $ modify $ \(p, _) -> (p, w)
```

Figure 5.18: The `MonadWeight` library for weighted random distribution.

where the weights of different functors are no longer derived by cardinality, but maintained by a list of doubles in the reader. Additionally, in the state monad, the integer stands for a pointer that locates the index of the current functor in the weight list. And the `Double` element stores an *accumulative weight* for calculation, embedded here for code simplicity.

Figure 5.18 defines the `MonadWeight` type class for auxiliary functions. Note that functors will be composed in a right-associative way, and this forms a one-to-one correspondence with the weight list. Hence the value of the pointer, say `i`, denotes the weight of the `i`th-functor. `pointer` gets the current pointer, and there are two valid operations to handle the pointer: `succ` moves the pointer one step to the right (i.e. adding the index by 1); `reset` resets the pointer to 0, and also resets the accumulative weight to 0 at the same time. `getWeight` gets a weight from the list specified by index. Then `getAcc` and `setAcc` are the getter and setter for the accumulative weight value, respectively. All those functions are provided with default implementations for `WeightT`.

Possible Failure Both Section 5.1.2 and Section 5.3.3 have shown the requirement for a representation of possible failure in generators. While Section 5.3.3 uses the composition of `Maybe` with a functor, it is more general to use the `MaybeT` transformer. Furthermore, although `empty` can be used to denote a failure, it is better to define a class specialized for `MaybeT`:

```
class Monad m => MonadMaybe m where
  none :: m a

instance Monad m => MonadMaybe (MaybeT m) where
  none = MaybeT $ return Nothing
```

Strategy: Weighted Distribution on Successful Generators. Finally, we synthesize randomness, weighted random distribution and possible failure in the instantiation of the monad:

```
(|*|_w) :: Cardinality g =>
    CoAlgM Weighted f a -> CoAlgM Weighted g a -> CoAlgM Weighted (f ⊕ g) a
(|*|_w) cF cG a = MaybeT $ do
  p  <- pointer
  rF <- runMaybeT $ fmap Inl (cF a)
  rG <- runMaybeT $ succ >> fmap Inr (cG a)
  when (getCard cG == 1 && isJust rG) (getWeight (p + 1) >>= setAcc)
  wF <- getWeight p
  wG <- getAcc
  case (rF, rG) of
    (Just _, Just _) -> setAcc (wF + wG) >> binomial (wF, return rF)
                                                     (wG, return rG)
    (Just _, _     ) -> setAcc wF >> return rF
    (_     , Just _) -> return rG
    _                -> return Nothing

resetW :: Weighted a -> Weighted a
resetW x = reset >> x
```

Figure 5.19: The combinator for weighted random distribution and the reset function.

```
type Weighted = MaybeT (WeightT (RandT Identity))
```

The desired strategy is to perform a weighted random distribution on only successful generators. This complicated behavior is encapsulated by two combinators, shown in Figure 5.19. Note that `|*|`$_w$ is the coalgebra combinator derived from the natural transformation `weightedTrafo`, shown in Appendix B.1. The derivation is straightforward, thus is omitted here.

There are two pre-assumptions: functors are composed by right-associative co-products; and the length of the weight list equals the number of functors, ensuring a one-to-one correspondence in order. Hence in `|*|`$_w$, `f` is always a single functor, but `g` can be a composite functor. As a result, the weight of `f` can be indexed from the weight list, while the weight of `g` has to be maintained in a chained coalgebra composition, using the *accumulative weight* in `WeightT`.

In Figure 5.19, `rF` is the result of the `f`-generator, and `rG` is obtained by running `cG` before the pointer is shifted by `succ`. Noticeable is that the weight of `g` is not the sum of all weights in `g`, but the total weight of *successful* generators in `g`. This invariant is maintained by `getAcc` and `setAcc`. Another point is that we need to check the boundary: when `g` is a single functor (i.e., with cardinality 1), the accumulative weight is exactly the weight of `g` when it succeeds.

Then it comes the pattern-matching on `rF` and `rG`. By our strategy, a successful generator is prior to a failing one, and if both succeed, the accumulator integrates both weights, and a random selection is performed between `rF` and `rG` using `binomial`. By this localized binary composition, the weighted distribution can be achieved globally for all functors in each round.

Finally, since `unfoldM` generates data by recursion, there are multiple rounds of random selection. Thus it is necessary to reset the state before each round. In Figure 5.19, this is done by `resetW`. The essence of `resetW` is again a natural transformation, but it is only specialized for the monad, and will

be applied in the middle of P1 and P2 (in Figure 5.12). Specifically, the construction would be:

```
unfoldM (unComp . out) . unfold (Comp . resetW . unCom . out)
                       . unfold (weightedTrafo . out)
                       . unfold (cF |*|_w cG |*|_w ...)
```

While after deforestation, it becomes `unfoldM (resetW . (cF |*|_w cG |*|_w ...))`.

RUN A GENERATOR *SCCL* runs a generator with weighted random distribution as follows:

```
generateW :: (Cardinality f, Traversable f) =>
    [Double] -> a -> CoAlgM Weighted f a -> IO (Fix f)
generateW ws input coalg = assert (getCard coalg == length ws) $
                           newStdGen >>= runWeighted
  where runWeighted g = return . fromJust . runIdentity . evalRandT g .
                        evalWeightT (0, 0) ws . runMaybeT $
                        unfoldM (resetW . coalg) input
```

An assertion checks if the length of the weight list is equal to the cardinality of `f` (i.e. the number of functors involved), and `coalg` is expected to be a composition of coalgebras with $|*|_w$.

5.5.2.3 Dynamic Distribution with Size Bound

SCCL further defines a specialized combinator for random construction with a dynamic distribution strategy. Additionally, the construction can be restricted with a size bound.

DYNAMIC DISTRIBUTION The dynamic distribution is designed as follows: the weights of functors are assigned at each round, based on their *arities* and the input. Functors with larger arities (i.e. more branches/sub-nodes in the constructors) are more likely to be selected in early rounds, so as to expand the tree; while functors with smaller arities tend to be selected when the tree goes deeper, so as to converge the generation quickly. To detect the depth of generation, we rely on the input to give such information. For example, the input type may contain a height size, which starts with a large number and is gradually decreased by coalgebras.

For simplicity, our weight function samples a binomial distribution. It is known that in a binomial distribution, the probability of having k successes in m independent trials is

$$prob(k, m, p) = \binom{m}{k} p^k (1-p)^{m-k}$$

Let `maxArity` be the maximum arity among functors, n be the current input, and N be the maximum input (namely the initial input). The weight of functor `f` at this stage is given by

$$W(\texttt{f}) = prob(\texttt{arity(f)}, \texttt{maxArity}, \frac{1+n}{2+N}) \tag{5.1}$$

where `arity(f)` denotes its arity. That is to say, at each round, functors are only differentiated in weight by their arities. During the process of generation, n gets smaller and smaller.

The arity class is defined in Figure 5.20, similar to `Cardinality`. The arity of a composite functor is defined as the maximum of arities of its members. Although it is possible to automatically derive the arity of a functor by Template Haskell [Adams and DuBuisson, 2012], currently it requires a manual assignment.

```
class Arity (f :: * -> *) where
  arity :: Proxy f -> Int

instance (Arity f, Arity g) => Arity (f ⊕ g) where
  arity _ = max (arity (Proxy :: Proxy f)) (arity (Proxy :: Proxy g))

getArity :: Arity f => CoAlgM m f a -> Int
getArity = arity . proxy
```

Figure 5.20: The arity of a functor.

SIZE BOUND The dynamic distribution can be used in situations where coalgebras do not enforce a convergence, and helps the generation to terminate. In that case, however, it is still possible to get into an infinite loop, when weights are not assigned in a good way. To enforce a termination, we add a strong bound on the number of recursions/constructions. It is exactly an upper bound for the number of constructors in random generation.

STRATEGY: DYNAMIC DISTRIBUTION ON SUCCESSFUL GENERATORS, WITH A SIZE BOUND. By defining type synonyms,

```
type MaxArity = Int
type MaxDepth = Int
type ReadEnv = (MaxArity, MaxDepth)

type Acc = Double
type Bound = Int
type StateEnv = (Acc, Bound)
```

the monad is directly constructed as

```
type Dynamic = MaybeT (StateT StateEnv (ReaderT ReadEnv (RandT Identity)))
```

including possible failure, dynamic distribution and size bound. The maximum arity (`MaxArity`) and the maximum/initial input (`MaxDepth`) are stored in a reader, while `Bound` is maintained in a state for checking size boundary. `Acc` is again an accumulator of weights.

In order to make our specialized combinator reusable, the carrier type is not directly instantiated to `Int`. Rather, it requires an `Int` to be included in the input:

```
class Derive a b | a -> b where
  derive :: a -> b
```

as expressed by `Derive a Int`.

The specialized combinators are presented in Figure 5.21. Note that $|*|_d$ is based on a natural transformation; whereas it requires a generalization of the product to store input values (as discussed in Section 5.3.3), thus the derivation steps are omitted here.

In Figure 5.21, the definition of $|*|_d$ follows a similar style to $|*|_w$: the generators could possibly fail, and the composition will skip the failing generators, but accumulate the weight of successful

```
(|*|_d) :: (Cardinality g, Arity f, Arity g, Derive a Int) =>
    CoAlgM Dynamic f a -> CoAlgM Dynamic g a -> CoAlgM Dynamic (f ⊕ g) a
(|*|_d) cF cG a = MaybeT $ do
  env <- ask
  rF <- runMaybeT $ fmap Inl (cF a)
  rG <- runMaybeT $ fmap Inr (cG a)
  when (getCard cG == 1 && isJust rG) $ setAcc $ weight env (arityG, thisDepth)
  let wF = weight env (arityF, thisDepth)
  wG <- getAcc
  case (rF, rG) of
    (Just _, Just _) -> setAcc (wF + wG) >> binomial (wF, return rF)
                                                     (wG, return rG)
    (Just _, _     ) -> setAcc wF >> return rF
    (_     , Just _) -> return rG
    _                -> return Nothing
  where arityF    = getArity cF
        arityG    = getArity cG
        thisDepth = derive a
        setAcc s  = modify $ \(_, c) -> (s, c)
        getAcc    = get >>= return . fst

resetD :: Dynamic a -> Dynamic a
resetD x = do (_, c) <- get
              if c <= 0 then none
                        else put (0, c - 1) >> x
```

Figure 5.21: The combinator for dynamic distribution and the reset function.

ones. The weight of a functor is derived by the `weight` function defined in Appendix B.2, following Formula 5.1. Moreover, the constraint `"Derive a Int"` allows us to grasp the current depth information from the input. For each round, the `resetD` function not only resets the accumulator, but also decreases the size bound by 1, and enforces a failure when the limit is reached.

RUN A GENERATOR A `Dynamic` generator can be executed in `IO`, by the following function:

```
generateD :: (Arity f, Traversable f, Derive a Int) =>
    Int -> a -> CoAlgM Dynamic f a -> IO (Maybe (Fix f))
generateD bound input coalg = newStdGen >>= runDynamic
  where maxArity = getArity coalg
        maxDepth = derive input
        env      = (maxArity, maxDepth)
        initS    = (0, bound)
        runDynamic g  = return . runIdentity . evalRandT g .
                        flip runReaderT env . flip evalStateT initS .
                        runMaybeT $ unfoldM (resetD . coalg) input
```

where `generateD` takes three parameters: the size bound, the input, and the coalgebra (in the form of `"cF |*|d cG |*|d ..."`. Compared with `generateW` which allows a runtime exception if the generator fails, `generateD` catches possible failure by `Maybe` in the return type.

5.5.3 Application II: Small-Step Evaluation

Another interesting application is the small-step evaluation of domain-specific languages. The inspiration comes from [Hutton, 1998] which illustrates that operational semantics are connected with the unfolding to transition trees, although their technique does not achieve modularity. While big-step semantics can easily be modularized by composing visitors or algebras, modularizing small-step semantics by modularizing coalgebras is non-trivial.

Compared with [Liang et al., 1995], for simplicity, *SCCL* only supports deterministic semantic rules with error handling, but it is already able to realize simple interpreters (shown later). Moreover, one can extend the current framework to further allow environments, non-determinism and so on by composing monad transformers.

The Priority Monad Transformer One big challenge of modularizing small-step evaluation is *ordering*. To tackle this, the central mechanism of our approach is to adopt priority-based composition.

```
newtype PriorT s e m a = PriorT {
  unPriorT :: MaybeT (StateT s (ExceptT e m)) a
}
```

The above code presents the priority monad transformer, where possible failure, priority state, and exception are embedded. The interface for such a transformer is shown in Figure 5.22, together with the default implementations for `PriorT`. `MonadPrior` firstly requires the priority type to be an instance of `Ord` for supporting comparison. Then it has two constructors: `create` produces a monadic value labelled with its priority; `failure` raises a failure. More importantly, the `priorComp` function realizes the priority-based composition strategy. The selection satisfies the following rules:

- The first exception (if any) will be thrown;
- Successes are prior to failures;
- For two successful results, the one with higher priority is preferred;
- If both results have the same priority, an exception will be thrown based on the priority.

Besides, `catchFail` takes two monadic values `x` and `y`; it returns the result of `x` by default, and only executes `y` when `x` fails. That is to say, this combinator catches and handles possible failure of `x` (but not exceptions).

A Generalization of Anamorphism For small-step evaluation, an `f`-coalgebra represents a set of transition relations related to functor `f`. The carrier represents the expression to be evaluated. However, we realize that the original coalgebra type `a -> m (f a)` is usually insufficient to model transition rules. Very often we need the more generalized form of coalgebra type shown below:

```
class (Monad m, Ord s) => MonadPrior s e m | m -> s, m -> e where
  create    :: s -> a -> m a
  failure   :: m a
  priorComp :: (s -> e) -> m a -> m a -> m a
  catchFail :: m a -> m a -> m a

instance (Monad m, Ord s) => MonadPrior s e (PriorT s e m) where
  create s a = PriorT $ put s >> return a
  failure    = PriorT . MaybeT $ return Nothing
  priorComp f x y = PriorT $ MaybeT $ do
    s <- get
    rX <- runMaybeT $ unPriorT x
    sX <- get
    put s
    rY <- runMaybeT $ unPriorT y
    sY <- get
    case (rX, rY, compare sX sY) of
      (Just _, Just _, GT) -> put sX >> return rX
      (Just _, Just _, LT) -> put sY >> return rY
      (Just _, Just _, EQ) -> throwError $ f sX
      (Just _, _     , _ ) -> put sX >> return rX
      (_     , Just _, _ ) -> put sY >> return rY
      _                    -> return Nothing
  catchFail x y = PriorT . MaybeT $ do
      r <- runMaybeT $ unPriorT x
      case r of Just _  -> return r
                Nothing -> runMaybeT $ unPriorT y
```

Figure 5.22: The `MonadPrior` interface for priority-based composition.

```
type CoAlgP g m f a = a -> m (Either (Fix g) (f (Either (Fix g) a)))
```

`CoAlgP` is closely related to *apomorphisms*, originally introduced by Vene and Uustalu [1998] as a generalization of anamorphism. Here the coalgebra has a more complicated form; as a consequence, it not only allows complete structures to be directly returned by coalgebras, but also allows fixpoints to appear at the sub-nodes of `f` to replace seeds. In the above definition, `f` is the functor of the coalgebra, while `g` denotes the final sums-of-product functor by composition. Hence the carrier type is usually equal to `Fix g`, but for genericity `a` remains abstract.

Correspondingly, the generic apomorphism replaces `unfoldM` with `unfoldP`:

```
unfoldP :: (Traversable f, Monad m) => CoAlgP f m f a -> a -> m (Fix f)
unfoldP f = join . fmap (h . fmap (fmap In . sequence . fmap (h . fmap (
    unfoldP f)))) . f
  where h :: Monad m => Either (Fix f) (m (Fix f)) -> m (Fix f)
        h (Left x) = return x
```

```
(|<>|) :: CoAlgP h Eval f a -> CoAlgP h Eval g a -> CoAlgP h Eval (f ⊕ g) a
cF |<>| cG = \x -> priorComp report (fmap (fmap Inl) (cF x))
                                    (fmap (fmap Inr) (cG x))
  where report :: Bool -> Conflict
        report True  = ConflictRdc
        report False = ConflictCgr

check :: CoAlgP f Eval f (Fix f) -> CoAlgP f Eval f (Fix f)
check coalg x = do
  isDone <- get
  if isDone then returnX
            else (put True >> coalg x) `catchFail` (put False >> returnX)
  where returnX = return (Left x)
```

Figure 5.23: The combinator for small-step semantics and the check function.

```
h (Right x) = x
```

STRATEGY: SINGLE-STEP DETERMINISTIC REDUCTION. To achieve modularization on small-step semantics, a minimal instantiation of the monad can be:

```
type Eval = StateT Done (PriorT IsRdc Conflict Identity)
```

given that

```
type Done = Bool
type IsRdc = Bool
data Conflict = ConflictRdc | ConflictCgr
```

Here `Done` represents a flag that indicates whether a single-step reduction is done. The priority is instantiated to `IsRdc`, where `True` is the label for *reduction rules*, and in contrast `False` for *congruence rules*. Moreover, `Conflict` captures two kinds of exceptions: conflicting reduction, and conflicting congruence. In fact, in a deterministic small-step semantic system, any expression cannot satisfy two reduction patterns or two congruence patterns at the same time, otherwise it is considered ill-formed. The only case with two patterns is one congruence and one reduction, where the latter is prioritized.

The composition strategy is implemented in Figure 5.23. Again, we define the coalgebra combinator, but omit the derivation from natural transformations, because it requires more generalization in the theory. The combinator `|<>|` immediately delegates the composition task to `priorComp`. Besides, `check` is the key function that guarantees the reduction to be *single-step*. If it observes the `Done` flag is `True`, the evaluation will be terminated immediately by returning the current seed. Otherwise the coalgebra will be applied to the seed; and if a failure is caught in the end, it implies that no progress can be made, for which the input is again returned. Since we might be executing a sub-branch of the whole recursion, a failure should not stop us from reducing other branches, therefore, the `Done` flag should be maintained carefully by the putters.

RUN AN EVALUATOR The following function runs an evaluator in the `IO` enviroment:

```
reduce :: Traversable f => Fix f -> CoAlgP f Eval f (Fix f) -> IO (Fix f)
reduce e coalg = case res of
  Left x          -> error (show x)
  Right (Just e') -> return e'
  Right _         -> return e
  where res = runIdentity . runExceptT . flip evalStateT False . runMaybeT .
              unPriorT . flip evalStateT False $ unfoldP (check coalg) e
```

where `coalg` has the form `"cF |<>| cG |<>| ..."` integrating all semantic rules.

EXAMPLE Chapter 3 of [Pierce, 2002] shows a simple untyped language representing arithmetic and boolean expressions. This can be captured in Haskell with the following two functors:

```
data ArithF a = TmZero | TmSucc a | TmPred a | TmIsZero a
data BoolF  a = TmTrue | TmFalse  | TmIf a a a
```

The semantic rules are represented by coalgebras of `ArithF` and `BoolF`, as shown in Figure 5.24. Note that *pattern synonyms* and *view patterns*, as extensions of GHC, are used in pattern matching to reduce boilerplate; `proj` denotes the *projection* operation to inspect the real functor from a co-product, dual to *injections* (`inj`); they are both defined under `:<:`, which denotes the membership of a functor to a co-product, as in DTC [Swierstra, 2008]. The `IsNumericVal` class with its `isNum` function, and `rdcRule` and `cgrRule` for labelling reduction/congruence rules are presented in Appendix B.3.

Smart constructors defined in Appendix B.4 as DTC can build the expression `"if false then 0 else (iszero (pred (succ 0)))"` as follows:

```
e :: Fix (ArithF ⊕ BoolF)
e = ifC false zero . iszero . pred $ succ zero
```

A single-step reduction on `e` is tested in GHCi:

```
> reduce e (evalArith |<>| evalBool)
iszero (pred (succ (zero)))
```

Where pretty-printing is omitted. By repeating `reduce` multiple times, we obtain the complete evaluation steps:

```
   (if false then 0 else (iszero (pred (succ 0))))
=> (iszero (pred (succ 0)))
=> (iszero 0)
=> true
```

5.5.4 Application III: Monadic Parsing

SCCL also involves simple functionality for modular monadic parsing, by defining parsers as coalgebras and composing them with the choice combinator. The inspiration comes from [Hutton and Meijer, 1998] which deals with parsing by effects. Although the current library only captures the fundamental features of parsing as a minimal core of Parsec [Leijen and Meijer, 2001], our goal is to reveal the strategy of selection behind the choice combinator, and also to understand modular parsing as an application of modular unfolds.

```
pattern Zero        <- (proj . out -> Just TmZero)
pattern Succ e      <- (proj . out -> Just (TmSucc e))
pattern Pred e      <- (proj . out -> Just (TmPred e))
pattern IsZero e    <- (proj . out -> Just (TmIsZero e))
pattern If e1 e2 e3 <- (proj . out -> Just (TmIf e1 e2 e3))
pattern T           <- (proj . out -> Just TmTrue)
pattern F           <- (proj . out -> Just TmFalse)

evalArith :: (ArithF :<: f, MonadPrior IsRdc Conflict m, IsNumericVal f) =>
    CoAlgP f m ArithF (Fix f)
evalArith (Pred Zero)              = rdcRule . Right $ TmZero
evalArith (Pred (Succ e)) | isNum e = rdcRule . Left $ e
evalArith (Pred e)                 = cgrRule . Right $ TmPred (Right e)
evalArith (Succ e)                 = cgrRule . Right $ TmSucc (Right e)
evalArith (IsZero e)               = cgrRule . Right $ TmIsZero (Right e)
evalArith _                        = failure

evalBool :: (ArithF :<: f, BoolF :<: f, MonadPrior IsRdc Conflict m,
    IsNumericVal f) => CoAlgP f m BoolF (Fix f)
evalBool (IsZero Zero)              = rdcRule . Right $ TmTrue
evalBool (IsZero (Succ e)) | isNum e = rdcRule . Right $ TmFalse
evalBool (If T x y)                 = rdcRule . Left $ x
evalBool (If F x y)                 = rdcRule . Left $ y
evalBool (If x y z)                 = cgrRule . Right $ TmIf (Right x) (Left y)
                                                             (Left z)
evalBool _                          = failure
```

Figure 5.24: Coalgebras of `ArithF` and `BoolF` representing small-step semantic rules.

THE PARSER MONAD TRANSFORMER *SCCL* defines `ParserT` as a lightweight transformer as follows:

```
newtype ParserT m a = ParserT { unParserT :: MaybeT (StateT String m) a }
```

where the state denotes an input string, and `MaybeT` for possible failure.

Next comes the lightweight `MonadParser` library that collects some parser combinators for use, as shown in Figure 5.25. Among them, the `choice` combinator is the key to the composition. While the `<|>` in Parsec only tries the second alternative if the first one fails without any consumption on the input (and hence requires `try` for manual backtracking), here `choice` automatically recovers the input on that failure.

STRATEGY: FIRST SUCCESSFUL PARSER. The coalgebra combinator straightforwardly invokes `choice` for alternative composition with automatic backtracking. The result comes from the first successful parser.

```
type P = ParserT Identity
```

```
class Monad m => MonadParser m where
  choice :: m a -> m a -> m a
  failP :: m a
  fetch :: m Char
  sat :: (Char -> Bool) -> m Char
  sat p = fetch >>= \x -> if p x then return x else failP
  char :: Char -> m Char
  char c = sat (== c)
  string :: String -> m String
  string ""     = return ""
  string (c:cs) = char c >> string cs >> return (c:cs)
  many :: m a -> m [a]
  many p = many1 p <|> return []
  many1 :: m a -> m [a]
  many1 p = p >>= \x -> many p >>= \xs -> return (x:xs)
  int :: m Int
  int = fmap read . many1 $ sat isDigit

instance Monad m => MonadParser (ParserT m) where
  fetch = ParserT $ get >>= \s -> if empty s then none
    else put (tail s) >> return (head s)
  failP = ParserT none
  choice x y = ParserT $ MaybeT $ do
    s <- get
    resX <- runMaybeT (unParserT x)
    case resX of
      Just _  -> return resX
      Nothing -> put s >> runMaybeT (unParserT y)
```

Figure 5.25: The lightweight `MonadParser` library.

```
(|<<|) :: CoAlgM P f a -> CoAlgM P g a -> CoAlgM P (f ⊕ g) a
(|<<|) cF cG a = fmap Inl (cF a) `choice` fmap Inr (cG a)
```

EXAMPLE We illustrate by implementing a simple parser for the following grammar:

```
expr ::= <int> | <int> '+' expr
```

which is represented by two functors and two coalgebras:

```
data FLit a = FLit Int
data FAdd a = FAdd Int a

pLit :: CoAlgM P FLit ()
pLit _ = liftM FLit int
```

```
pAdd :: CoAlgM P FAdd ()
pAdd _ = do n <- int
            char '+'
            return $ FAdd n ()
```

Try the composite parser in GHCi:

```
> let parse s = runIdentity . flip evalStateT s . runMaybeT . unParserT $
    unfoldM (pAdd |<<| pLit) ()
> parse "1+2+3"
Just (1 + (2 + 3))
```

POSSIBLE REFINEMENTS Currently `ParserT` is still restricted in terms of practicality, partly due to the algorithmic challenges of modularizing parsing we have discussed in Section 4.1.1. On the other hand, some more generalized recursion schemes need to be involved for improvements. Consider the following concrete syntax for let-expressions:

```
e ::= ... | "let" <string> "=" e "in" e
```

for which we might define its abstract syntax as follows:

```
data LetF x = Let String x x    deriving (Functor, Foldable, Traversable)
```

However, the derived `Traversable` instance is not desirable, because `"in"` has to be parsed in the middle of effects. This problem is related to a customized evaluation order of effects. While [Pardo, 1998] avoids this issue by dealing with every concrete syntax in the datatype definition, an alternative approach is to generalize coalgebras into *Mendler-style* [Mendler, 1991; Uustalu and Vene, 2000] coalgebras. With effects, such a coalgebra has type

```
type MCoAlgM m f a = forall x. (a -> m x) -> a -> m (f x)
```

Note that the construction on `f` is still top-down, but effect handling can be more flexible, in a similar way to open recursion.

Other possibilities to improve the library includes introducing new specialized combinators, for example with longest-match, or with memoization to make efficient backtracking and potentially to achieve left recursion, but we leave those for our future work.

5.6 Summary

This chapter explores the composability of producer operations by composing coalgebras in functional programming, and presents *SCCL* for practical modularity in random generators, small-step semantics, and monadic parsing. Consequently, production strategies are untangled with composition strategies, making each part interchangeable and the composition easier. By using monad transformers, coalgebras can be reused in different contexts.

The library has foundations in some theoretical exploration, where producers are naturally composed to produce product forests, and later they are transformed into sums-of-products with natural transformations. A deforestation shows that it is equivalent to use specialized coalgebra combinators that encapsulate certain selection strategies.

Chapter 6

Case Study: Random Generators and Enumerators

To illustrate the utility of *SCCL*, and evaluate the selection strategies behind the specialized coalgebra combinators, this chapter presents a case study on modularizing random generators and enumerators. In the case study, random generators and enumerators are represented by monadic `f`-coalgebras, and they allow a diversity of combinations between carrier types and composition strategies flexibly.

6.1 Overview

The purpose of our case study is to illustrate that with *SCCL*, we can modularize data generators on modular ASTs. Meanwhile, by untangling generation behaviors with composition strategies, we obtain great flexibility and code reuse. Consequently client code can be simplified using specialized coalgebra combinators. To do so our experiments have involved 9 language constructs with different kinds of generators, and evaluated on the distribution of those constructs in generated structures, to validate the effectiveness of *SCCL* combinators. On the other hand, our experiments have also evaluated the code size and execution time compared with non-modular approaches using QuickCheck, to show how much code reuse and performance impact such modularization brings.

The case study shows that with reasonable performance penalty, generators can be designed as modular and reusable components, and the use of combinators frees users from implementing ad-hoc composition algorithms, enhancing code conciseness and utility. Moreover, the case study shows an additional benefit of using *SCCL*: custom random generators can be ported to QuickCheck for checking properties.

Language Constructs The famous book "Types and Programming Languages" (TAPL) [Pierce, 2002] has been used in Chapter 4 as well as in other work [Zhang and Oliveira, 2017, 2018] for case studies, because it collects a lot of language features, and by gradually introducing them it shows an evolution of interpreters. Hence TAPL is a good material for evaluating modularization techniques. This case study only focuses on a small subset of TAPL languages. Logically the 9 constructs originate from three different features: arithmetic expressions (numeric literals, addition and multiplication);

booleans (boolean literals, conditional and equality); and lambdas (variables, abstraction and application). They are defined as functors:

```
-- arith
data LitF   e = Lit Int
data AddF   e = Add e e
data MulF   e = Mul e e

-- bool
data BoolF  e = BoolV Bool
data IfF    e = If e e e
data EqualF e = Equal e e

-- lam
data VarF   e = Var Int
data LamF   e = Lam Type e
data AppF   e = App e e

type LNG = LitF ⊕ AddF ⊕ MulF ⊕ BoolF ⊕ IfF ⊕ EqualF ⊕ VarF ⊕ LamF ⊕ AppF
```

Those functors are composed by the right-associative sum (⊕) into a whole functor (`LNG`). Thus `Fix LNG` denotes the type of expressions using those data variants. Notice that `VarF` represents variables with *de Bruijn indices* [De Bruijn, 1972]. `LamF` models typed lambda constructs, where `Type` consists of integer, boolean and function types. To avoid mutual recursion, `Type` is defined in a non-extensible way:

```
data Type = TLit | TBool | TFunc Type Type
```

To assist in our experiments and evaluation, a few valid operations are implemented as algebras, for example: pretty-printing, calculating tree height, checking variable binding (whether there are free variables), and collecting occurrences of different constructs in depth. Furthermore, a simple type checker and an interpreter are implemented for `Fix LNG`, using recursive functions; potentially it is possible to write them in an extensible style, but it is out of our focus. Finally, `Arity` instances are manually written for the functors, but it is expected that the arities can be derived by templates in the future.

STRATEGIES Traditionally, the implementation of a generator entangles the strategy of generation with the composition strategy. To differentiate them:

- *Seed type*: Seeds specify the requirements and guide the generation process. In non-modular generators, seeds are merely the arguments passed among recursive functions. In the context of unfolds, a seed refers to the *carrier* of a coalgebra. In particular, most generators are specified with a size in the seed type.
- *Composition*: Combinators reflect certain strategies of generator composition. For random generators, a combinator usually models a probability distribution among the candidates.

In the case study, there are various kinds of properties for seed types, including *constructor size* (CS), *depth size* (DS), *bound variables* (BV), *bound variable types* (BVT), *expected type* (ET), and

integer set (IS). On the other hand, we select three strategies of coalgebra composition from the *SCCL* library, namely *weighted distribution* (WD) and *dynamic distribution* (DD) for random distribution, and *merging results* (MG) for enumeration. Those seed types and composition strategies are reused in different experiments, achieving significant modularity.

Research Questions Our experiments have been conducted for answering the following research questions:

RQ1: Can we indeed modularize the implementation of generators for different constructs? Can we untangle the generation strategies from composition strategies, making both as reusable components? Is there any preference in different composition strategies?

RQ2: Can we have a modular random generator that distributes constructors in a relatively fair way?

RQ3: Can we have a modular random generator that distributes constructors in a dynamic way during construction?

RQ4: How do random generators associate with QuickCheck for testing properties?

RQ5: How much does such modularization contribute to code size reduction? How much is the impact on performance?

6.2 Random Generators as Coalgebras

Random generators are represented by (monadic) coalgebras. In order to generate well-formed `Fix LNG` expressions on a particular size, the carrier type can be:

```
type Size  = Int
type MaxId = Int
type Seed  = (Size, MaxId)
```

where `Seed` stands for the carrier type. `MaxId` records the maximum number of bound variables by lambdas, to ensure that no free variables are generated. However, `Size` can have different interpretations. Normally, the size may express the total number of constructors (nodes) in a tree, or the tree height/depth.

Constructor Size More precisely, we treat constructor sizes as 0-based, hence it actually represents the number of *internal nodes* of a tree. For instance, generators of `LitF` and `AddF` are implemented as follows:

```
gLit :: (MonadMaybe m, MonadRand m) => CoAlgM m LitF Seed
gLit = liftS $ \n -> if n > 0 then none else do
  x <- choose (0, 100)
  return $ Lit x

gAdd :: (MonadMaybe m, MonadRand m) => CoAlgM m AddF Seed
gAdd = liftS $ \n -> if n <= 0 then none else do
```

```
    n' <- choose (0, n - 1)
    return $ Add n' (n - 1 - n')

liftS :: (Functor f, Functor m) => CoAlgM m f Size -> CoAlgM m f Seed
liftS c (size, max) = fmap (fmap (\x -> (x, max))) (c size)
```

For literals and additions, since `MaxId` is not used, an auxiliary function `liftS` helps to pass the `MaxId` argument with no change, therefore "lifting" a `Size` coalgebra into a `Seed` coalgebra. In `gLit` , when the input size is non-positive, an integer is randomly picked between `[0, 100]`; otherwise `gLit` raises a failure. In contrast, `gAdd` merely accepts positive sizes; a size is randomly divided into two smaller ones for the sub-expressions. Also notice that both generators are generic on monad `m`, which ensures the reusability of these coalgebras for different composition strategies.

For lambda expressions, a parameter type is generated by a random pick from `TLit`, `TBool` and a unary `TFunc` for simplicity.

Automated Composition of Coalgebras Similarly to DTC [Swierstra, 2008], we encapsulate generator coalgebras with type classes, in order to obtain automatic composition. Two composition strategies of *SCCL* are adopted here: the `Weighted` monad for a weighted distribution, and `Dynamic` for a dynamic distribution on the input size and the arities of data constructs. For instance, `Weighted` coalgebras are captured by the following class:

```
class Traversable f => RandomW f a where
  genW :: CoAlgM Weight f a

instance (RandomW f a, RandomW g a, Cardinality g) => RandomW (f ⊕ g) a
  where genW = genW |*|_w genW
```

$|*|_w$ is the weighted distribution combinator in *SCCL*. After implementing all `RandomW` instances for the 9 functors, the coalgebras are automatically composed. Finally, *SCCL* offers `generateW` to build a runnable random generator under the `IO` environment. By initializing the weights equally, a uniform distribution is achieved on successful generators.

```
testW :: Size -> IO (Fix LNG)
testW n = generateW (replicate 9 1) (n,-1) (genW :: CoAlgM Weighted LNG Seed)
```

Similarly, to perform the dynamic distribution strategy with $|*|_d$, we define `RandomD`:

```
class Traversable f => RandomD f a where
  genD :: CoAlgM Dynamic f a

instance (RandomD f a, RandomD g a, Cardinality g, Arity f, Arity g,
    Derive a Int) => RandomD (f ⊕ g) a where
  genD = genD |*|_d genD
```

and again make the coalgebras as instances of `RandomD`. Although this introduces some boilerplate, the coalgebras are doubtlessly reused. Likewise, to test the generator in `IO`:

```
testD :: Size -> IO (Maybe (Fix LNG))
testD n = generateD 10000 (n, -1) (genD :: CoAlgM Dynamic LNG Seed)
```

where `10000` is the size bound.

Table 6.1: The distribution of different constructors in 10000-round tests, showing the average number of occurrences. For constructor size, the input is 100. For depth size the input is 10.

Seed	Composition	Avg. #Occurrences of Constructors								
		Lit	BoolV	Var	Add	Mul	If	Equal	Lam	App
CS + BV	WD	38.9	38.9	23.1	16.6	16.7	16.6	16.7	16.7	16.7
DS + BV	WD	26.1	26.2	12.1	10.5	10.6	10.5	10.6	10.6	10.6

Depth Size When the input represents the depth of construction, the number of constructors will increase exponentially along with the input. The carrier type remains the same, but the coalgebras shall be implemented differently. Taking `AddF` for example:

```
gAdd' :: (MonadMaybe m, MonadRand m) => CoAlgM m AddF Seed
gAdd' = liftS $ \n -> if n <= 0 then none else do
  n' <- choose (0, n - 1)
  (nA, nB) <- elements [(n - 1, n'), (n', n - 1)]
  return $ Add nA nB
```

The depth size is again 0-based. Different from `gAdd`, `gAdd'` takes input size `n`, but passes `n - 1` to either its left or right child, and gives the other child a random but smaller size. This ensures that the depth is decreased only by 1 at a time. Coalgebras defined on the depth size can be composed in the same way as those on the constructor size.

Answer to RQ2 We have implemented both constructor-size and depth-size generators for all the functors. To evaluate the effectiveness of uniform distribution, both generators are combined by $|*|_w$ (with cardinality-based weights), and 10000-based testing is performed on them respectively, which collects the average number of occurrences of each functor. Table 6.1 shows the results of the constructor-size generator (with input 100) and the depth-size generator (with input 10). The uniform distribution differentiates internal nodes from leaf nodes, and generally the constructors are distributed in a relatively fair way. The 6 constructs on the right appear to have the same frequency. And `Var` differs from `Lit` and `BoolV`, due to the restriction on variable binding.

Answer to RQ3 To also evaluate the dynamic distribution strategy in *SCCL*, the generators are instead composed by $|*|_d$ (binomial distribution), and we measure the average depth of the occurrences of each constructor. Here depth means the distance from the constructor node to the root of the tree. In Table 6.2, it is observed that constructs with different arities are clearly distinguished in depth: while `If` is generated mostly near to the root, `Lit`, `BoolV` and `Var` generally have a larger depth during construction.

Answer to RQ1 The experiments have indicated great modularity and reusability in client code. Firstly, functors and co-products modularize the AST and the random generation. Secondly, coalgebras can be reused for different composition strategies, because of the monad transformers; on the other hand, the combinators are also reusable for different coalgebras, due to the genericity in the carrier type.

Table 6.3 shows the distribution of constructors with $|*|_d$. One can notice that `Lam` greatly dominates other constructs in terms of frequency. The reason is that, both constructor-size and depth-size

Table 6.2: The distribution of different constructors in 10000-round tests, showing the average depth of occurrences. For constructor size, the input is 100. For depth size the input is 10. Size bound 10000.

Seed	Composition	Avg. Depth of Constructor Nodes								
		Lit	BoolV	Var	Add	Mul	If	Equal	Lam	App
CS + BV	DD	10.3	10.3	10.7	6.5	6.5	0.9	6.5	8.4	6.5
DS + BV	DD	7.0	7.0	7.7	5.0	5.0	2.9	5.0	5.9	5.0

Table 6.3: The distribution of different constructors in 10000-round tests, showing the average number of occurrences. For constructor size, the input is 100. For depth size the input is 10. Size bound 10000.

Seed	Composition	Avg. #Occurrences of Constructors								
		Lit	BoolV	Var	Add	Mul	If	Equal	Lam	App
CS + BV	DD	9.1	9.0	8.6	5.8	5.7	1.4	5.8	75.6	5.7
DS + BV	DD	17.2	17.3	9.3	9.4	9.4	2.7	9.4	24.9	9.3

generators generate trees of exactly the given size. Thus during a construction, whenever the size is great than 0, only internal constructs can be selected, of which `Lam` has the least arity: 1. This indicates that for random generation that sticks to a specific size, the weighted distribution performs better in terms of randomness and diversity.

6.3 Generating Well-Typed Expressions

The above generators can efficiently produce a large number of *well-formed* expressions in seconds. Nevertheless, there is only a small portion of them that can be type-checked for use. Worse still, the percentage plunges when the size is increased, as shown in Table 6.4. To counter this problem and enhance the utility, it is necessary to implement generators that only build *well-typed* expressions. There has already been some existing work on generating well-typed expressions [Claessen et al., 2014; Fetscher et al., 2015] efficiently. Our case study realizes a simple generator by maintaining the expected type of term in the carrier type:

```
type Size       = Int
type Env        = [Type]
type ExpectType = Maybe Type
type TypedSeed  = (Size, ExpectType, Env)
```

`ExpectType` is an optional `Type`, where `Nothing` implies that any type is acceptable. Besides, for lambda expressions, the types of bound variables are maintained in a list (`Env`). `Size` here stands for the depth size. Below is the `AddF` generator:

```
gTypedAdd :: (MonadMaybe m, MonadRand m) => CoAlgM m AddF TypedSeed
gTypedAdd = liftS $ \(n, t) ->
  if t `notElem` [Just TLit, Nothing] then none else do
    let n0 = max 0 (n - 1)
    n' <- choose (0, n0)
    (nA, nB) <- elements [(n0, n'), (n', n0)]
```

Table 6.4: The percentage of well-typed expressions generated with (CS + BV, WD) in 100000-based testing.

Size	# Well-Typed	% Well-Typed
0	100000	100.00%
1	37479	37.48%
2	10873	10.87%
3	2898	2.90%
4	741	0.74%
5	163	0.16%
6	60	0.06%
7	14	≈ 0%

```
return $ Add (nA, Just TLit) (nB, Just TLit)
```

Since an addition can only be typed as `TLit`, the expected input type should either be `Just TLit` or `Nothing` as uninitialized. In fact, it is tricky to generate a well-typed expression on a particular size, because it can easily fall into a dilemma, where either nothing is generated, or the tree becomes too large or even infinite. To tackle this issue, the size in the carrier is merely a weak bound. As shown above, `gAdd` accepts the case when `n = 0`. Two smaller seeds are assigned to the children, where both are further required with `TLit`. Besides the weak bound, `Dynamic` maintains a strong "size bound" on the number of constructors to prevent infinite loops. More importantly, the convergence of random generation comes from the dynamic distribution: when the depth size gets smaller (i.e. the tree goes deeper), the constructors with smaller arities (such as `Lit`, `BoolV` and `Var`) are more likely to be selected and converge the generation quickly.

ANSWER TO RQ1 The well-typed expression generators have been tested with both $|*|_w$ and $|*|_d$ respectively. However, our experiments show that $|*|_w$ with a uniform distribution can easily get into infinite loops, since non-leaf constructs have a majority and tree nodes are expanded endlessly. If we intentionally increase the weights of `Lit`, `BoolV` and `Var`, the trees will be mostly generated with only one node.

Table 6.5 presents the result of a 10000-based generation with seed type (DS + BVT + ET), and strategy DD ($|*|_d$), taking 10 as the depth input and 10000 as the size bound. Also note that among the 10000 generated trees, the average height is 10.6. The distribution of different constructs looks reasonable. Note that such a generator can possibly generate a one-node tree, or even a tree with hundreds of nodes. But when the input size is increased, the probability distribution becomes more smooth, and the height of the generated tree tends to get closer to the ideal value. In contrast, an input of less than 5 makes the generator easy to reach the size bound, where nothing is generated. Overall, we consider the dynamic distribution strategy to be better for generating well-typed expressions, because it offers better control over congruence.

Table 6.5: The distribution of different constructors in a 10000-round well-typed expression generation, with input 10.

	Lit	BoolV	Var	Add	Mul	If	Equal	Lam	App
Avg. #Constrs	18.7	10.9	4.5	4.3	4.3	4.1	3.5	18.8	12.8
Avg. Depth	6.3	5.9	7.6	4.7	4.6	2.9	4.0	6.1	4.6

6.4 Enumerating Expressions

A third strategy of generation, which also appears in many testing frameworks [Duregård, 2012], are enumeration-based generators. Our case study enumerates expressions of a specified depth size into a list. The carrier type of coalgebras becomes:

```
type Size     = Int
type MaxId    = Int
type IntSet   = [Int]
type EnumSeed = (Size, MaxId, IntSet)
```

`EnumSeed` is composed of three parts: the depth size (DS), the number of bound variables (BV), and additionally a finite integer set (IS) for `Lit` to choose from. The `AddF` enumerator is shown below for illustration:

```
eAdd :: CoAlgM [] AddF Seed
eAdd = liftSE eAdd'
  where eAdd' n = if n <= 0 then [] else [Add x y | (x, y) <- pairs n]
        pairs n = (n-1, n-1) : concat [[(x, n-1), (n-1, x)] | x <- [1..n-2]]

liftSE :: (Functor f, Functor m) => CoAlgM m f Size -> CoAlgM m f Seed
liftSE c (size, max, set) = fmap (fmap (\x -> (x, max, set))) (c size)
```

The monad is instantiated to `[]`. The auxiliary function `liftSE` allows `eAdd` to focus only on the input size. When the size `n` is positive, it enumerates all possible pairs with maximum depth `n - 1`, and correspondingly generates `Add` expressions.

Note that, for even relatively small sizes, the number of elements in the enumeration list is very large. However, since efficiency is beyond our scope, we have not applied optimizations. One possibility of improvement is to make a variant of `unfoldM` and apply memoization/sharing to avoid duplicate calculations.

6.5 Checking Properties with QuickCheck

The integration into QuickCheck enhances the utility of our generators. As a consequence, we have packed our code into QuickCheck generators, and further defined a few properties and validated them using QuickCheck. More specifically, we have encoded:

- the correctness of deforestation (Corollary 6) from a product forest to a fixpoint of sums-of-products, showing the equivalence among the three approaches:

  - the unfold-transformation pattern based on a natural transformation;
  - a single unfold based on the natural transformation and the general coalgebra combinator;
  - a single unfold based on a specialized coalgebra combinator.

  The code is based on the well-typed generators of `LitF` and `AddF` with a weighted (uniform) random distribution.

- the type-preservation of evaluation on the combined language (`Fix LNG`), stating that for any well-typed expression, the evaluation does not change its type;
- the property that evaluating well-typed `Fix LNG` terms does not introduce free variables.

DEFORESTATION The three approaches correspond to the following functions respectively:

```
genA :: TypedSeed -> Weighted (Fix (LitF ⊕ AddF))
genA = unfoldM (unComp . out) . unfold (Comp . resetW . unComp . out) .
    unfold (weightedTrafo . out) . unfold (g1 <**> g2)
  where (g1, g2) = (Comp . gTypedLit, Comp . gTypedAdd)

genB :: TypedSeed -> Weighted (Fix (LitF ⊕ AddF))
genB = unfoldM (resetW . unComp . weightedTrafo . (cF <**> cG))
  where (g1, g2) = (Comp . gTypedLit, Comp . gTypedAdd)

genC :: TypedSeed -> Weighted (Fix (LitF ⊕ AddF))
genC = unfoldM (resetW . (gTypedLit |*|_w gTypedAdd))
```

Recall that `<**>` is the general coalgebra combinator. `weightedTrafo` and `resetW` were introduced in Section 5.5.2.2; the former realizes weighted random selection as a natural transformation, and the latter updates the monad context in each round. The property for checking correctness of deforestation is presented below:

```
deforest :: Int -> Int -> Bool
deforest n g = rA == rB && rB == rC
  where n' = n `mod` 10
        g' = mkStdGen g
        rA = runGen genA
        rB = runGen genB
        rC = runGen genC
        runGen :: (TypedSeed -> Weighted a) -> Maybe a
        runGen gen = runIdentity . evalRandT g' . evalWeightT (0,0) [1,1] .
                     runMaybeT $ gen (n', Nothing, [])
```

Here `n` gives the input size, and is rescaled into the range $[0, 10)$. `g` creates a `StdGen` random seed for the generators, and to ensure its randomness, the range of `Int` generator should be large enough.

CUSTOMIZED QUICKCHECK GENERATOR Now suppose that we have a well-typed expression generator of functor `f`. To transform this generator into a QuickCheck generator, we first define the following datatype:

```
data MaybeExpr f = MaybeExpr { unMaybe :: Maybe (Fix f) }
```

The reason for such a definition is to avoid the conflict with QuickCheck's pre-defined instances.

Next we are able to integrate a generic generator into QuickCheck as follows:

```
instance (Traversable f, Arity f, RandomD f T.TypedSeed) =>
    Arbitrary (MaybeExpr f) where
  arbitrary = do
    n <- getSize
    seed <- chooseAny
    let coalg = genD :: CoAlgM Dynamic f TypedSeed
    let res   = unfoldM (resetD . coalg) (n, Nothing, [])
    let env   = (getArity coalg, n)
    let initS = (0, 1000)
    let g     = mkStdGen seed
    return . MaybeExpr . runIdentity . evalRandT g . flip runReaderT env
           . flip evalStateT initS . runMaybeT $ res
```

with the aforementioned `resetD` and `getArity` in Section 5.5.2.3. It defines a custom generator based on the dynamic distribution strategy, and is able to produce `Fix f` if the size bound `1000` is not exceeded.

ANSWER TO RQ4 Now we are able to define the aforementioned second and third properties in QuickCheck, and test them with our own random generators:

```
preserveType :: MaybeExpr LNG -> Property
preserveType (MaybeExpr expr) = isJust expr ==> isJust t && t == t'
  where e  = fromJust expr
        t  = typeCheck e
        t' = typeCheck (evaluate e)

bounded :: MaybeExpr LNG -> Property
bounded (MaybeExpr expr) = isJust expr ==> isBounded (evaluate e)
  where e = fromJust expr
```

Details about type-checking (`typeCheck`), evaluation (`evaluate`) and checking the variable binding (`isBounded`) are presented in Appendix C. Finally, all three properties have been tested by QuickCheck:

```
> quickCheckWith (stdArgs {maxSize = 1000000}) deforest
+++ OK, passed 100 tests.
> quickCheckWith (stdArgs {maxSize = 30}) preserveType
+++ OK, passed 100 tests.
> quickCheckWith (stdArgs {maxSize = 30}) bounded
+++ OK, passed 100 tests.
```

Table 6.6: SLOC of non-modular (QuickCheck) code vs modular (*SCCL*) code, on implementing constructor/depth-size generators with weighted/dynamic distribution.

SLOC	AST + aux	CS-coalgs	DS-coalgs	CS + WD	CS + DD	DS + WD	DS + DD	*Total*
Non-Modular	49	-	-	51	51	67	67	285
Modular	21	42	42	3	3	3	3	117

Table 6.7: SLOC of non-modular (QuickCheck) code vs modular (*SCCL*) code, on implementing *arith*, *bool* and *lam* generators.

SLOC	AST			Coalgs	Generators			*Total*
	arith	bool	lam		arith	arith + bool	arith + bool + lam	
Non-Modular	3	6	9	-	17	31	52	118
Modular	10			42	3	3	3	61

6.6 Evaluation: Code Size and Execution Time

Answer to RQ5 To have a rough estimation on how much our modularization techniques reduce the generator code, we have implemented two simple benchmarks on the absolute number of source lines of code (SLOC, lines of code without counting empty lines, comments and imports). Table 6.6 compares the non-modular (QuickCheck) code with the modular (*SCCL*) code, when implementing constructor-size and depth-size generators for the 9 constructs with both weighted distribution and dynamic distribution. The non-modular code requires more auxiliary functions to be defined for realizing those distributions. It is observed that the reusability of both coalgebras and combinators reduces the glue code significantly with *SCCL*. On the other hand, to illustrate the modularity of data variants, we define the language AST in an extensible style: *arith* (3 constructs), *arith* + *bool* (6 constructs) and finally *arith* + *bool* + *lam* (9 constructs). Table 6.7 shows the comparison of SLOC on implementing constructor-size weight-distributed generators. Again, since both the AST and the generators are reusable, there is a considerable reduction in the modular approach.

To also have a rough estimation on how much performance penalty the modularization brings, we conduct an experiment on the contrast between the well-typed generator (DS, DD) and a non-modular QuickCheck generator by measuring the execution time. The QuickCheck generator is implemented in a conventional style, using mutually recursive functions, and builds random well-typed terms of a non-extensible datatype with the 9 constructors.

Our benchmark programs were executed on a Thinkpad with 2.70 GHz Intel Core i5-5257U and 8GB RAM. The execution time was measured by the *Criterion*[1] library in Haskell. Table 6.8 shows the mean execution time of *SCCL* and QC (QuickCheck) per round on 5 different sizes: 10, 15, 20, 25 and 30. In those cases, *SCCL* is approximately 1.5x - 2x slower than QC. There are three possible aspects for the distinction. Firstly, the modularity of abstract syntax tree and random generators with *SCCL* makes the program naturally less efficient, because there are more function calls (like injections) and intermediate results during composition. Although the QuickCheck version takes less execution time, neither the AST nor the generators are extensible. Secondly, the recursion patterns are different in both implementations. *SCCL* uses coalgebras and hence follows a top-down strategy, while QC makes use of general recursion functions that behave as a mixture of top-down and

[1] `https://github.com/bos/criterion`. criterion-1.5.3.0. Bryan O'Sullivan.

Table 6.8: Mean execution time of *SCCL* approach and QuickCheck per round on 5 different input sizes.

Size	Exec. Time of *SCCL* (DS, DD) (ms)	Exec. Time of QC (ms)	Ratio
10	2.469	0.967	2.55
15	6.130	2.095	2.93
20	9.448	3.860	2.45
25	18.310	6.076	3.01
30	25.440	10.060	2.53

bottom-up manners. Last but not least, the dynamic distribution is performed on *successful* results, and in *SCCL* coalgebras and random selection run in parallel; while it is tricky to implement this in QC, and instead the probabilistic distribution generates a random permutation in advance, before coalgebras are composed by the permutation in each round. Similarly, this permutation mechanism is tricky to implement in *SCCL*, for it requires a "global view" of the functors, while our framework has the foundation in binary composition. It could be interesting to explore the MRM-style [Oliveira et al., 2015] using list-of-functors as an alternative in the future.

Limitations There are more insights about the limitations and possible optimizations. At present, *SCCL* generators are difficult to realize *backtracking*, and it may require a more generalized recursion scheme. On the other hand, we can do optimizations by fusing the process of generation with subsequent procedures (such as type-checking and evaluation) in practice, by existing fold/unfold fusion laws for deforestation. Finally, it is observed that there is still boilerplate code that can be reduced or avoided, such as the definition of `RandomW` and `RandomD` specialized for monads.

Chapter 7

Related Work

This chapter captures the most relevant work in various areas we have touched, and includes some discussion as well as comparison with those techniques.

7.1 Design Patterns for Extensibility and Modularity

Algebra of Programming (AoP) Bird and de Moor [1997] present a mathematical framework based on a categorical calculus of relations for calculating programs. As a result, algorithmic strategies can be formulated abstractly without reference to specific datatypes, furthermore, derivations and proofs can be conducted in a principled way. More specifically, AoP captures the abstract shape of datatypes by *functors*, and operations (traversals) by *algebras* and generic recursion schemes like *folds* (*catamorphisms*). Consequently, certain relations and some theorems can be observed from category theory [Herrlich and Strecker, 1973]. Although the theory has been well studied in a functional context, it had limited impact in practical programming until [Swierstra, 2008] which reveals an elegant solution to the Expression Problem.

OO Design Patterns The famous "Gang of Four" (GoF) presents design patterns [Gamma, 1995] as repeatable solutions to commonly occurring problems in software design. They provide templates to structure object-oriented code for modularity and reuse. Although design patterns have been pervasively used in software development, the main criticism might be that design patterns are not well abstracted and lack formal foundations, as a result they can be hard to be made as reusable components or libraries. Nevertheless, some programmers believe that they can actually be captured abstractly, but existing languages have limited features to support such abstractions. There has been some research [Gibbons, 2003a,b, 2006; Gibbons and Oliveira, 2009; Oliveira, 2007; Oliveira et al., 2008] on abstracting a few design patterns by higher-order recursive functions including `map`, `fold` and `unfold` in functional programming, and simultaneously they draw a connection to the Algebra of Programming [Bird and de Moor, 1997].

The Expression Problem Since Wadler [1998] firstly posted the example of an arithmetic language, the Expression Problem was identified to express the need for certain extensibility in program design. Specifically, it requires an existing framework to be extensible on both data variants and

operations, and meanwhile ensures type-safety, leading to *separate compilation* and *modular type-checking*. So far there have been quite a few different solutions proposed in various paradigms. Torgersen [2004] presented four different solutions applicable to Java and/or C#, using advanced language features including type parameters with constraints, *F-bounded polymorphism* [Canning et al., 1989], *wildcards* [Torgersen et al., 2004], and runtime reification of type parameters. Sadly those approaches either sacrifice static type safety, or require large amounts of boilerplate code on types to ensure such safety. Later Odersky and Zenger [2005a] proposed two families of solutions in Scala using *abstract type members* and *nested classes*. One family is based on object-oriented decomposition, while the other on functional decomposition using *visitors*. As a result, both solutions can not only support consumers that merely collect information on trees, but also *binary methods* and tree transformations. Main criticism on that work can be that it highly relies on the language features and requires significant boilerplate composition code. Wang and Oliveira [2016] proposed a simple OO solution that gets rid of type parametrization, and only uses *covariant type refinements*. In that case, however, the extensibility becomes restricted on binary methods. More closely related to the thesis are the solutions including EMGM [Oliveira et al., 2006], Finally Tagless [Carette et al., 2007], Data Types à la Carte [Swierstra, 2008] and Object Algebras [Oliveira and Cook, 2012a] discussed below.

Data Types à la Carte (DTC) Swierstra [2008] proposed his DTC approach in Haskell to demonstrate that AoP [Bird and de Moor, 1997] concepts can address the Expression Problem. It uses functors and generic folds, with type-level co-products (sums) for extensibility. With fixpoints, data structures are reified into real objects, hence are easily accessible via pattern matching. However, DTC suffers from expressiveness and practicality: while DTC can model recursive tree-like structures with functors, more sophisticated mutually recursive structures, and even generalized algebraic datatypes (GADTs) [Schrijvers et al., 2009], are difficult to model in that way. Later Bahr and Hvitved [2011] introduced more recursion schemes and enhanced the expressiveness of computations by realizing *monadic catamorphisms* and contextual *term homomorphisms* upon DTC; they further adopted *higher-order functors* [Johann and Ghani, 2008] to represent mutually recursively datatypes and GADTs. Oliveira et al. [2015] promoted the work in a different direction, by replacing binary co-products with type-level list-of-functors, and presents the MRM framework for extensible and reusable pattern matching. Our thesis, specifically Chapter 5, makes use of the original DTC framework to represent data structures; therefore the same limitations as DTC apply. While computations on data structures in existing work are more related to *consumers* and *transformations*, the key novelty of Chapter 5 is the exploration on *builders* and the modularity behind (monadic) *anamorphisms*.

Church Encodings, Finally Tagless and Object Algebras The idea originates in [Hinze, 2006] which shows that Haskell type classes can represent encodings of datatypes. Later Oliveira et al. [2006] applied some variations to the encodings and showed that their approach gives a solution to the Expression Problem. Later Finally Tagless (or Tagless Final) [Carette et al., 2007] popularized the technique and showed how the interpretation of simple embedded DSLs can be realized in functional programming; the approach is discussed more in detail in [Kiselyov, 2012] on extensibility, and compositional operations with context. Finally Object Algebras [Oliveira and Cook, 2012a] generalized the idea to object-oriented programming [Oliveira and Cook, 2012a] and revealed its great practicality. They all have the foundation in Church encodings [Böhm and Berarducci, 1985; Church, 1936].

In object-oriented programming, Object Algebras are closely related to *internal visitors* [Buchlovsky

and Thielecke, 2006; Oliveira et al., 2008]. The relationship between internal visitors and Church encodings was first demonstrated theoretically by Buchlovsky and Thielecke [2006]. In contrast with DTC [Swierstra, 2008], Object Algebras naturally allow mutually recursive datatypes using multiple type parameters; however reification of data structures is sacrificed, instead generic functions are used to represent objects. Later Oliveira et al. [2013] proposed a generic and type-safe composition mechanism for Object Algebras, and meanwhile generalized Object Algebras to model *external visitors* [Buchlovsky and Thielecke, 2006; Oliveira et al., 2008] based on *Parigot encoding* [Parigot, 1992]. More recently Rendel et al. [2014] extended Object Algebras and built the connection to *attribute grammars* [Knuth, 1968]. Cazzola and Vacchi [2016] proposed a trait-based approach to modularize the abstract syntax and semantics of DSLs; the technique is very similar to Object Algebras, but uses Scala traits with abstract types instead of parametric polymorphism.

Unfortunately, those techniques had not been discussed in related work for traversal boilerplate, modular parsing, and the composability of co-algebras. Although this thesis does not explore new solutions to the EP, it builds on top of the existing techniques for addressing the first two problems. More precisely, chapters 3 and 4 adopt the original Object Algebras pattern as the underlying framework, therefore inheriting its merits in terms of type-safe extensibility, modularity and conciseness. For example in Chapter 4, with the goal of semantic modular parsing, we chose Object Algebras for representing data, because the pattern is relatively lightweight and makes good use of existing OO features, such as inheritance, generics and subtyping. It coexists well with Packrat parsing [Ford, 2002] in Scala, and models mutually recursive (multi-sorted) languages expressively, leading to the concise code throughout the chapter. Furthermore, our work in those chapters can be generalized by introducing new encodings or patterns. Recently after the original publication of Chapter 3, Zhang and Oliveira [2017] used the generalized Parigot-encoded Object Algebras to implement an extensible Java Visitor framework with flexible traversal control.

7.2 Modularity of Operations in Functional Programming

Our thesis specifically focuses on two types of operations: *consumers*, which basically traverse and consume tree structures and collect information; and *producers*, which dually build data structures. Regarding categorical recursion schemes, consumers and producers usually refer to *catamorphisms* (folds) and *anamorphisms* (unfolds) [Meijer et al., 1991] in functional programming, respectively.

Modularity of Folds Folds and their modularity have motivated plenty of research in the area of functional programming. Swierstra [2008] in his DTC approach shows several techniques for developing datatypes and their corresponding operations modularly. The operations are modularly defined using `f`-algebras (and some `f`-algebra variants). The composition of `f`-algebras is automated using type classes, but in its essence the automated composition uses the generic algebra combinator:

```
(<+>) :: (f a -> a) -> (g a -> a) -> (f ⊕ g) a -> a
(<+>) aF _  (Inl x) = aF x
(<+>) _  aG (Inr x) = aG x
```

aforementioned in Chapter 5. Later Bahr and Hvitved [2011] generalized DTC with higher-order functors, but the algebra combinator only requires slight changes on `(<+>)`. Oliveira et al. [2015] used

list-of-functors for a replacement of binary-sum functors in their MRM framework, where algebras are composed by `(:::)` into a list, and then `extractAt` extracts the exact algebra during a fold.

Object Algebras are closely related to folds and `f`-algebras. Although the representation is different from the more conventional sums-of-products with `f`-algebras, Buchlovsky and Thielecke [2006] have clearly demonstrated the isomorphism, where a sum type corresponds to the composition (product) of visitors in the *Visitor* pattern, and further in Church encodings. The product of visitors/algebras is simply achieved in Object Algebras using OO inheritance, resulting in similar modularity benefits as DTC.

Modularity of Unfolds As discussed in Chapter 5, although it is widely acknowledged that a generic coalgebra combinator leads to the product of functors based on type isomorphisms from the theory, there has been unfortunately little work putting this into practice, partly due to the lack of examples or applications on modularizing coalgebras. There are indeed interesting applications of producers/builders with modularity in sight, including parsing, interpretation and so on, yet most existing work tackles them with modularizing folds, general recursions or even syntactic techniques without type-safety. In DTC, Swierstra [2008] mentioned that "*While we have encountered the fold over a data type, I have not mentioned the unfold*", later Bahr and Hvitved [2011] built on his work and mentioned the composition of a *term homomorphism* and a *cv-coalgebra* [Vene, 2000] resulting in another *cv-coalgebra*, which is used for a more general recursion scheme called futumorphism [Vene, 2000]. However the composition of two coalgebras is again not mentioned. Oliveira et al. [2015] proposed MRM as a variant of DTC using list-of-functors, but discussed only `f`-algebras and fold-like operations.

In Chapter 4, we achieve type-safe modular parsing in an object-oriented context, by relying on the natural composability of algebras, and more importantly on the parser combinators. In Chapter 5, our approach manages to compose modular parsers and semantic components by specialized coalgebra combinators in *SCCL*. More detailed literature review on modular parsing and modular semantics lies in Section 7.4 and Section 7.5, respectively.

Modularity in Theory Before AoP [Bird and de Moor, 1997] involved categorical concepts in programming, the properties of functors and recursion schemes have been well studied in category theory [Herrlich and Strecker, 1973]. Since Gill et al. [1993] proposed the idea of *shortcut fusion* in Haskell code, general *fusion laws* on recursion schemes including catamorphisms, anamorphisms and hylomorphisms were further presented in [Hinze et al., 2011; Onoue et al., 1997; Pardo, 1998; Takano and Meijer, 1995]. They offer theoretical foundations for program reasoning and optimization in terms of deforestation [Wadler, 1988].

An important remark is that the sense in which most of those papers use "modularity" for unfolds is different from ours in Chapter 5. Mostly modularity refers to the fusion laws on anamorphisms, including the *ana-fusion law* [Pardo, 1998]:

```
g . f = fmap f . g' ⇒ unfold g . f = unfold g'
```

which combines a cohomomorphism with an anamorphism to achieve another anamorphism. Furthermore, in the general monadic case, the ana-fusion law becomes:

```
g . f = fmap (fmap f) . g' ⇒ unfoldM g . f = unfoldM g'
```

In our work, modularity means the ability to develop `f`-coalgebras for two different functors independently and later being able to combine them into a single coalgebra. But there is some relationship: the above laws have been used in proving Theorems 3 and 5 respectively. Besides, Theorems 1 and 3 have been presented in [Hinze et al., 2011] and [Hinze, 2010] respectively.

7.3 Structure-Shy Traversals

This section discusses some related work on boilerplate code in various traversal patterns and the concept of structure shyness, specifically for Chapter 3.

ADAPTIVE OBJECT-ORIENTED PROGRAMMING (AOOP) AOOP is an extension of object-oriented programming aimed at increasing the flexibility and maintainability of programs [Lieberherr, 1996]. AOOP promotes the idea of *structure-shyness* to achieve those goals. In AOOP it is possible to select parts of a structure that should be visited. This is useful to do traversals on complex structures and focus only on the interesting parts of the structure relevant for computing the final output. The original approach to AOOP was based on a domain-specific language [Lieberherr, 1996]. DJ is an implementation of AOOP in Java using reflection [Orleans and Lieberherr, 2001]. More recently DemeterF [Chadwick and Lieberherr, 2010] improved previous approaches to AOOP by providing support for type-safe traversals, generics and data-generic function generation. *Shy* shares with AOOP the use of structure-shyness as a means to increase flexibility and adaptability of programs. Most AOOP approaches, however, are not type-safe. The exception is DemeterF where a custom type system was designed to ensure type-safety of generic functions. Unlike DemeterF, which is a separate language, *Shy* is a Java library. Moreover, the compilation of DemeterF programs is implemented through static weaving, and thus appears to preclude separate compilation.

STRATEGIC PROGRAMMING Strategic programming is an approach to data structure traversal, which originated in *term rewriting*. Visser et al. [1998] extended the rewriting strategies of [Borovanský et al., 1996] with generic one-level traversal operators enabling a style of term rewriting where computations are represented by simple, conditional rewrite rules, but the application of such rules is controlled separately using the concept of a strategy. Strategies can be primitive (e.g., "fail") or composed using combinators (e.g., "try s else s'"). These and other combinators were formalized in a core language for strategic rewriting in [Visser and Benaissa, 1998a].

The strategy concept has since then been ported to other paradigms. JJTRAVELER is an OO framework for strategic programming [Visser, 2001b]. Lämmel and Visser [2002] introduced typed strategy combinators in Haskell. The relation between strategic programming and AOOP has been explored in [Lämmel et al., 2003].

The key tenet of strategic programming is separation of concerns: actual computation and traversal are specified separately. In *Shy*, the traversal of a data structure is also specified separately (in a super-interface), however, it is fixed for specific styles of queries and transformations. For instance, both queries and transformations employ an innermost, bottom-up strategy.

The distinction between queries and transformations also originates from existing work in strategic programming. Lämmel and Visser [2002] discuss *type unifying* and *type preserving* traversals. Type unifying traversals correspond to queries, where all data type constructors are unified into a sin-

```
interface AccuTrafoExp<M, E> extends ExpAlg<Pair<M, E>> {
   Monoid<M> m();
   ExpAlg<E> expAlg();
   default Pair<M, E> Var(String s) {
      return new Pair<>(m().empty(), expAlg().Var(s));
   }
   default Pair<M, E> Lit(int i) {
      return new Pair<>(m().empty(), expAlg().Lit(i));
   }
   default Pair<M, E> Add(Pair<M, E> e1, Pair<M, E> e2) {
      return new Pair<>(m().join(e1.fst(), e2.fst()),
                        expAlg().Add(e1.snd(), e2.snd()));
   }
}
```

Figure 7.1: Combining query and transformation.

gle monoid. Analogously, *Shy* transformations are type preserving in the sense that a transformation is an algebra which maps constructor calls to another algebra of the same type.

The ASF+SDF program transformation system distinguishes transforming and accumulating traversals, which correspond to our transformations and queries, respectively [van den Brand et al., 2003a]. Furthermore, an *accumulating transforming* traversal combines both styles, by tupling the accumulated result and the transformed tree. This combination could easily be generated by *Shy* by having the boilerplate code construct the monoid and the transformed term in parallel (see Fig. 7.1).

STRUCTURE-SHY TRAVERSALS WITH VISITORS A standard way to remove boilerplate in OOP is to use *default visitors* [Nordberg III, 1996]. Default visitors can be used in similar ways to our generic traversals. Many programmers using the visitor pattern create such default visitor implementations to avoid boilerplate code. There are two important differences to our work in Chapter 3. Firstly conventional visitors are not extensible in a type-safe way. Secondly, with *Shy* the code for generic traversals is automatically generated, whereas with default visitors such code usually has to be implemented by hand.

Visser [2001b] adapted the strategy combinators of Stratego [Visser and Benaissa, 1998a; Visser et al., 1998] to combinators that operate on object-oriented visitors [Gamma, 1995]. The resulting framework JJTRAVELER solves the problem of entangling traversal control within the `accept` methods, or in the visitors themselves (which only allow static specialization). A challenge not addressed by JJTRAVELER is type safety of traversal code: either the combinators needs to be redefined for each data type, or client code needs to cast the generic objects of type `AnyVisitable` to the specific type. Even if specific combinators would be generated, however, the traversed data types would not be extensible.

Another approach to improve upon the standard *Visitor* pattern is presented in [Palsberg and Jay, 1998]. This work particularly addressed the fact that traditional visitors operate on a fixed set of classes. As a result, the data type can not be extended without changing all existing visitors as well. The proposed solution is a generic `Walkabout` class which accesses sub-components of arbitrary data

structures using reflection. Unfortunately, the heavy use of reflection makes `Walkabouts` significantly slower than traditional visitors. The authors state that the `Walkabout` class could be generated to improve performance, but note that the addition of a class could trigger regeneration. As a result the pattern does not support separate compilation. Our solution obtains the same kind of default behavior for traversal, without losing extensibility, type safety, or separate compilation.

Whereas the `Walkabout` provides generic navigation over an object structure, this navigation can be programmed explicitly using *guides* [Bravenboer and Visser, 2001]. Guides insert one level of indirection between recursing on the children of a node in `visit` methods: the guide decides how to proceed the traversal. Since guides need to define how to proceed for each type that will be visited, they suffer from the same extensibility problem as ordinary visitors. Generic guides, on the other hand, are dynamically typed and use reflection to call appropriate `visit` methods. The `Walkabout` can be formulated as such a generic guide.

Structure-Shy Traversals in Functional Programming In functional programming, there has been a lot of research on type-safe structure-shy traversals. Lämmel and Peyton Jones' "*Scrap your Boilerplate*" (SyB) [Lammel and Jones, 2003; Lämmel and Jones, 2004, 2005] series introduced a practical design pattern for doing generic traversals in Haskell. The simple queries and transformations in *Shy* were partly inspired by SyB. However SyB and *Shy* use very different implementation techniques. SyB is implemented in Haskell and relies on a run-time type-safe cast mechanism. This approach allows SyB traversals to be encoded once-and-forall using a single higher-order function called `gfoldl`. In contrast, in *Shy* Java annotations are used to generate generic traversals for each structure.

A drawback of SyB traversals is that they are notoriously slow, partly due to the use of the run-time cast [Adams and DuBuisson, 2012]. Another notable difference between SyB and *Shy* is with respect to extensibility. While *Shy* supports extensibility of both traversals and structures, the original SyB approach did not support any extensibility. Only in later work, Lämmel and Peyton Jones proposed an alternative design for SyB, based on type classes [Wadler and Blott, 1989]. This design supports extensibility of traversals, but not of the traversed structures.

Closest to *Shy* is an approach proposed by Lämmel et al. [2000] for dealing with the so-called "*large bananas*". A large banana corresponds to the *fold algebra* of a complex structure. Object Algebras, which we use in our work, are an OO encoding of fold algebras [Oliveira and Cook, 2012a; Oliveira et al., 2008]. However Lämmel et al. work has not dealt with extensibility. Interestingly in their future work, Lämmel et al. did mention that they would like "to cope with incomplete or extensible systems of datatypes".

Language Extensions for Queries Inspired by XPath/XQuery, there has been some work on adding support for similar types of queries on object-oriented structures. For example, the work on $C\omega$ [Bierman et al., 2005] extends C# with generalized member access, which allows simple XPath-like path expressions. Thus in $C\omega$ it is possible to express queries quite concisely. However, in contrast to *Shy*, $C\omega$ is a language extension and it does not deal with transformations.

Eliminating Boilerplate in Design Patterns Design patterns [Gamma, 1995] improve the design and modularity of object-oriented programs. However, the implementation of design patterns sometimes requires significant amounts of boilerplate code. There has been some work on imple-

menting design patterns in AspectJ to achieve reusability and modularity [Hannemann and Kiczales, 2002], and thus eliminate some of the boilerplate code. A challenge with traditional design patterns, however, is that the boilerplate code is not always mechanical, due to many possible implementation choices. Chapter 3 proposes a number of design patterns for traversals. Because implementing these design patterns by hand would be quite tedious, we automatically generate the code for such design patterns using *Shy*. Fortunately, in contrast to many of the traditional design patterns, the code for generic traversals is highly regular and easy to generate, and can be completely eliminated.

7.4 Modular Parsing

This section reviews some existing work on modular parsing, mostly for Chapter 4. Additionally there is some related work on modular parsing by anamorphisms, and we draw its connection to Chapter 5.

Safely Composable Type-Specific Languages There is almost no work on semantically modular parsing. A notable exception is the work on safely composable *type-specific languages* [Omar et al., 2014]. In this work, the extensible language Wyvern supports the addition of new syntax and semantics, while preserving type-safety and separate compilation. However this approach and our work in Chapter 4 have different goals: their approach is aimed at supporting extensibility of Wyvern with new syntax; whereas our approach is a general technique aimed at modular parsing of any languages. In contrast to their modular parsing approach, which is directly built-in to the Wyvern language, our approach is library-based and can be used by many mainstream OO languages.

Syntactically Extensible Parsing Extensible parser generators [Gouseti et al., 2014; Grimm, 2006; Parr and Quong, 1995; Schwerdfeger and Van Wyk, 2009a; Viera et al., 2012; Warth et al., 2016a] are a mainstream area of modular syntax and parsing. They allow users to write modular grammars, where new non-terminals and production rules can be introduced, some can even override existing rules in the old grammar modules. For instance, *Rats!* [Grimm, 2006] constructs its own module system for the collection of grammars, while NOA [Gouseti et al., 2014] uses Java annotation to collect all information before producing an ANTLR [Parr and Quong, 1995] grammar and the parsing code. Those parser generators focus on the *syntactic extensibility* of grammars: they rely on whole compilation to generate a global parser, even if there is only a slight modification in the grammar. Some of those parser generators may statically check the correctness and unambiguity of grammars. In contrast, because our approach is based on parser combinators, there is no support for ambiguity checking. However, as far as we are aware, no extensible parser generators support separate compilation or modular type-checking. It is worth mentioning that in [Viera et al., 2012], users can define grammar fragments as typed Haskell values, and combine them on the fly. Later they are processed by a typed parser generator. Nevertheless this requires a lot of advanced language features, making client complex. Our approach in Chapter 4 is simple and a straightforward use of OO programming, and makes parsing code directly reusable.

Macro systems like the C preprocessor, C++ templates and Racket [Tobin-Hochstadt et al., 2011], and other meta-programming techniques are a similar area aiming at syntactic extensibility. SugarJ [Erdweg et al., 2011] conveniently introduces syntactic sugar for Java using library imports. Composition of syntactic sugar is easy for users, but it requires many rounds of parsing and adaption, hence

significantly affects the efficiency of compilation. Since the implementation was based on SDF [Heering et al., 1989] and Stratego [Visser, 2001a], it does not support separate compilation. Racket adopts a macro system for library-based language extensibility [Tobin-Hochstadt et al., 2011]. It uses attributed ASTs for contextual information, and extensions can be integrated in a modular way. However such modularity is not flexible enough for language unification, as the syntax is only built from extensions. Extensible compilers like JastAdd [Ekman and Hedin, 2007] and Polyglot [Nystrom et al., 2003] also support extensible parsing, but it is mostly done using parser generators. They focus on the extensions to a host language. Those techniques are short of type safety in a modular setting as well.

Extensible Parsing Algorithms *Parse table composition* [Bravenboer and Visser, 2008; Schwerdfeger and Van Wyk, 2009b] is an approach where grammars are compiled to modular parse tables. Those parse tables are expressed as DFAs or NFAs, and later they can be composed by an algorithm, to provide separate compilation for parsing. The generation of parse tables can be quite expensive in terms of performance. The approach is quite different from ours, since it uses parse tables, whereas we use parser combinators. Our approach supports both separate compilation as well as modular type-checking, and is commonly applicable OO languages. Moreover, the extensibility of parsing is further available in language composition.

Parser Combinators Parser combinators have become more and more popular since [Burge, 1975; Wadler, 1985]. Many parsing libraries produce recursive descent parsers by introducing functional *monadic parser combinators* [Hutton and Meijer, 1996]. Parsec [Leijen and Meijer, 2001] is perhaps the most popular parser combinator library in this line. It is widely used in Haskell (with various "clones" in other languages) for context-sensitive grammars with infinite lookahead. Nevertheless, Parsec users suffer from manual *left-recursion elimination*, high cost for *backtracking* and *longest match composition* issues, as we discussed in Section 4.1.1. Those limitations make Parsec (and similar parsing techniques) inadequate for modular parsing.

Some recent work on parser combinators [Ford, 2002; Frost et al., 2008; Might et al., 2011] proposed a series of novel parsing techniques that address the issue of left-recursion. We chose Packrat parsing due to its simplicity in Scala, but in general there are alternatives to it.

Despite the algorithmic challenges, Pardo [1998] proposed the concept of *monadic anamorphisms* and presented monadic parsing as an application. The modular parsing in *SCCL* is closely related to his work, namely parsers are defined as monadic coalgebras, and composed with specialized coalgebra combinators. While Chapter 5 points out that it can be troublesome to parse *abstract syntax* due to ad-hoc traversals, Pardo [1998] avoids the issue by parsing *concrete syntax*, and further fuses parsing with subsequent semantic actions into *hylomorphisms*, to deforest abstract syntax trees. Unfortunately, encoding concrete syntax requires much more sorts in mutually recursive types, and complicates the code significantly. An alternative approach is generalizing coalgebras and anamorphisms with *Mendler-style* [Mendler, 1991; Uustalu and Vene, 2000] for explicit traversal control.

Case Classes Scala case classes can encode algebraic datatypes that allow the addition of new constructors. However such "open" case classes do not enforce exhaustiveness of pattern matching for extensible operations, and thus do not provide a full solution to the Expression Problem. Nevertheless case classes are widely used in practice, and a solution for parsing open case classes (and compos-

ing such parsers) is quite relevant in practice. The techniques in Section 4.2 can be readily adapted to work with case classes. The work by Sloane and Roberts [2015] on a modular Oberon compiler applied similar techniques with packrat parsers and case classes. In Chapter 4 we use Object Algebras for full extensibility and type safety, and we have well studied the algorithmic challenges of parsing in a modular setting.

7.5 Modular Semantics and Generators

This section lists a few related topics and some existing work on modular semantics and data generators, specifically for Chapter 5 and Chapter 6.

Modular Semantics Similar to modular parsing, related research on modular semantics has been conducted in various branches. There are syntactic modularization techniques prevalent in most language workbenches and extensible compilers, for example Rascal [Klint et al., 2009], SugarJ [Erdweg et al., 2011], JastAdd [Ekman and Hedin, 2007], and Polyglot [Nystrom et al., 2003]. They basically allow modular grammars/specifications in certain forms, and later apply code generation for language semantics. However, different from modular parsing, modular semantics have received considerable attention in related literature, specifically for *denotational semantics* [Scott and Strachey, 1971] and sometimes *big-step operational semantics*, because denotational semantics are normally designed with composability, and they can be implemented as *consumers* that are easily modular. As a result, many (partial) solutions to the Expression Problem have presented modular semantics as their applications. Modular visitors, and their generalizations/variants have been used in [Leduc et al., 2017; Oliveira, 2009]. In recent years, the rise of Object Algebras [Oliveira and Cook, 2012a] and similar patterns motivates modular semantics in [Biboudis et al., 2016; Inostroza and Storm, 2015; Warth et al., 2016b]. Oliveira et al. [2013] generalized Object Algebras with *object self-references* and *call-by-name parameters* for better traversal control in implementing semantics. More recently, Zhang and Oliveira [2017] have also adopted that generalized pattern, and especially implemented small-step semantics for TAPL interpreters, yet had to sacrifice modularity on ASTs. In functional programming, Swierstra [2008] presented modular denotational semantics with composable functors and algebras in DTC. Delaware et al. [2013] used Church encodings for extensible semantics, and invoked *Mendler-style algebras* [Uustalu and Vene, 2000] for explicit traversal control. Since Object Algebras are closely related to folds in functional programming, those techniques can be traced back to categorical representations.

Structured operational semantics [Plotkin, 1981], or *small-step semantics*, on the other hand, have been poorly explored using coalgebras. They are quite useful in inspecting step-by-step execution, and offer more convenience to program reasoning. In fact, there has already been well-established theory about operational semantics and their categorical representations [Lenisa et al., 2004; Turi and Plotkin, 1997]. Recently, Honsell et al. [2004] and Steingartner et al. [2016] explored some theory to represent small-step *transition relations* among states as coalgebras, and Hutton [1998] had presented some practical Haskell code for illustration earlier. However, those coalgebras are normally based on some fixed *observable behavior functors*, rather than the *signature functors* that reveals the abstract syntax of languages as in denotational semantics in order for modularity. [Jaskelioff et al., 2011] integrates both algebraic and coalgebraic features using both kinds of functors, nevertheless,

they still implement transition coalgebras on a behavior functor, and achieve modularity on signature functors by the generic algebra combinator in essence. Worse still, the code for semantics written by users is rather complicated.

Our attempt in Chapter 5 straightforwardly represents transitions as coalgebras with respect to the syntax functor, and designs specialized combinators in *SCCL*. The resulting client code is concise and immediately reflects the correspondence to small-step semantic specifications (rules). It is worth mentioning that modular semantics sometimes require a second dimension of modularity, namely on functionality, and we use *monad transformers* inspired by [Liang et al., 1995].

Generating Random Expressions Random testing is widely used in domain-specific language implementations for validating properties and finding bugs. QuickCheck [Claessen and Hughes, 2000] is a well-known Haskell library for random data generation and random testing. Our case study implements modular random generators by using coalgebras and unfold, and also associates with QuickCheck for validating properties.

There has been a lot of work on generating random expressions. A challenging problem is to deal with languages including lambdas and applications, when the goal is to generate well-formed expressions [Grygiel and Lescanne, 2013; Wang, 2005]. Moreover, there is usually a demand for generating well-typed expressions [Claessen et al., 2014; Fetscher et al., 2015; Grygiel and Lescanne, 2013; Pałka et al., 2011]. Besides, [Christiansen and Fischer, 2008; Duregård et al., 2012; Runciman et al., 2008] proposed various techniques for enumeration-based generation. Those methods are often generic to languages and rules, whereas our case study focuses more on the modularity of the generators and the use of specialized combinators. As far as we know there is no previous work on building modular generators for modular datatypes before.

Chapter 8

Conclusion

This chapter concludes the dissertation and points out the possible directions for future work.

8.1 Summary

This dissertation has argued that modularizing consumer and producer operations is an important and challenging area of today's modular programming. Following the Expression Problem [Wadler, 1998], we have identified several new research directions in both object-oriented programming and functional programming, and proposed our methodologies for tackling the corresponding issues. There are other practical aspects associated with modularization, including type-safety, conciseness, expressiveness and efficiency. Taking all those aspects into consideration, we have encoded our techniques into some libraries, and further presented applications as well as case studies to demonstrate their utility. More precisely:

- Chapter 3 identified the boilerplate issue in modularizing consumers of abstract syntax trees, and showed how various types of default traversals can be automatically provided by *Shy*. *Shy* traversals are written directly in Java and are type-safe, extensible and separately compilable. There has always been a tension between the correctness guarantees of static typing, and the flexibility of untyped/dynamically-typed approaches. *Shy* shows that even in type systems like Java's, it is possible to get considerable flexibility and adaptability for the problem of boilerplate code in traversals of complex structures, without giving up modular static typing. The case study on a domain-specific questionnaire language showed a significant reduction in code size.

- Chapter 4 focused on parsing, a representative of producer operations, and showed how type-safe modular parsing can be achieved in OO languages. The algorithmic challenges of modular parsing have been identified, and finally an integration of Packrat parsing, Object Algebras and OO inheritance forms our solution in Scala. As a result, not only parsers can evolve together with the abstract syntax, but also separate compilation and modular type-checking are guaranteed. The language feature abstraction further enhances code reuse and modularity. The TAPL case study demonstrates a great reduction in code size with reasonable performance penalty.

- Chapter 5 explored the modularity of producer methods in a more general view, and illustrated the essence of modular unfolds by a general composition operator for `f`-coalgebras that produce product structures. We have shown that taking the fixpoint of such product structures creates product forests, and a following transformation can be used to produce the more familiar sums-of-products. To avoid the intermediate product forests, deforestation theorems are presented for both pure and monadic coalgebras, when transformations are natural and unfold-based. Finally in practice, it is more convenient to use the derived specialized combinators on coalgebras to directly generate sums-of-products. The resulting *SCCL* library has integrated a couple of different selection strategies, and has contributed to the modularization in applications like random generators, small-step semantics and monadic parsing. In Chapter 6, the case study on random generators and enumerators reveals code reusability and flexibility in various dimensions.

8.2 Future Work

This section shows possible directions of future work, by discussing some limitations and deeper insights into the proposed methodologies.

Boilerplate and Code Generation There are some places to improve regarding boilerplate code generation. Currently *Shy* generates separate files for the generic traversal patterns; although they are put into different packages, those classes/interfaces are still polluting the namespace, and the connection to the annotated Object Algebra interfaces is unclear. It is possible to modify the abstract syntax tree, and inject the boilerplate code inside the Object Algebra interfaces for encapsulation; this can be done by using Lombok for example.

In Section 4.4.3, the manual composition of language components is verbose in `VarLamExpr`:

```
object VarLamExpr {
  trait Alg[E, T] extends VarExpr.Alg[E] with TypedLam.Alg[E, T]
  trait Parse[E, T] extends VarExpr.Parse[E] with TypedLam.Parse[E, T] {...}
  trait Print extends VarExpr.Print with TypedLam.Print
}
```

Such a pattern refers to *family polymorphism* [Ernst, 2001] for reducing boilerplate. We desire to write the following imaginary code for language composition:

```
object VarLamExpr[E, T] extends VarExpr[E] with TypedLam[E, T] {
  trait Parse {...}
}
```

where the language components and the type parameters are only specified at the outermost layer. `Alg`, `Parse` and `Print` are implicitly defined and the subtyping relations are automatically derived. Only additional code shows up in trait `Parse`. Such simplification can either be done by the compiler of the host language, or by using meta-programming techniques.

There are also some minor issues. In Section 5.5.2.3, currently the arity of a functor has to be assigned by users; potentially this information can be automatically derived by Template Haskell [Adams and DuBuisson, 2012] from datatype definitions.

Extension to Libraries The libraries proposed in this dissertation can be extended for more expressiveness. Currently *Shy* captures four kinds of traversal patterns, however, they are all in a bottom-up manner. It could be possible to extend *Shy* traversals with more flexible traversal strategies, similar to strategic programming [Borovanskỳ et al., 1996; van den Brand et al., 2003b; Visser and Benaissa, 1998b]. On the other hand, since Chapter 4 explores modular parsing and uses Object Algebras [Oliveira and Cook, 2012a], the *Shy* framework could also be extended to support the definition of parsers and more generalized producer operations.

For *SCCL*, a couple of selection strategies have been captured for specialized coalgebra composition, and common monads like reader, writer, state, maybe, and list have been involved. It would be interesting to explore new monads (such as `IO`) and their strategies in new applications.

Modular Semantics and Modular Monadic Parsing Small-step semantics and monadic parsing are two applications of modularizing producers in Chapter 5. They can be improved and extended in various ways. In modular semantics, we have only presented rules for simple constructs. If the language is extended, for example, with abstraction and application, the monad will have to be extended with a state transformer to maintain an environment for variables. Writing larger examples can give better illustration for its utility.

For modular monadic parsing, we have only presented a simple implementation of the choice combinator. In fact, memoization can be developed in the design of monads, and will offer great help in encoding left-recursive grammars and enhance the efficiency. On the other hand, custom evaluation of effects in parsing has also been discussed in Section 5.5.4, where *Mendler-style coalgebras* [Mendler, 1991; Uustalu and Vene, 2000] were proposed for a potential generalization.

(Functional) Language Workbenches Language workbenches aim to define and compose language components easily. Erdweg et al. [2015] has presented a feature model that captures the major features of language workbenches, including notation, semantics, editor support, validation, testing and composability. This dissertation has discussed the modularity of consumer and producer operations, without breaking the modularity of ASTs. Specifically, we can identify some language features as follows:

- *Consumers:* validation (like type-check), interpretation, transformation (like desugaring), ...
- *Producers:* parsing, interpretation, random test generation, ...

This motivates us to modularize language implementations at various aspects, and finally produce language workbenches for convenient language extension and composition. By realizing such modularization in functional programming with algebras, coalgebras, and the generic recursion schemes (folds, unfolds, ...), we can further apply some fusion theorems and deforestation techniques to enhance the performance.

Data Representations and Expressiveness The combinators in *SCCL* are implemented based on natural transformations over two functors. Such binary composition makes it tricky to encode the behaviors that require a global view of all functors (as discussed in Section 6.6). That inspires us to follow the MRM [Oliveira et al., 2015] framework, using the list-of-functor representation. The original MRM library has only encoded (generalized) algebras, as follows:

```
data Matches (fs :: [* -> *]) (a :: *) (b :: *) where
  Void :: Matches '[] a b
  (:::) :: Functor f => (f a -> b) -> Matches fs a b -> Matches (f ': fs) a b
```

Thereby we can possibly extend the library with (monadic) coalgebras, and realize the generic anamorphism together with specialized coalgebra combinators.

On the other hand, the expressiveness of functors in Haskell is still restricted due to the lack of mutual recursion. In contrast, Object Algebras [Oliveira and Cook, 2012a] make it straightforward to represent mutually recursive data structures. To achieve this in Haskell, *higher-order functors* [Johann and Ghani, 2008] could be a solution (also related to GADTs), but have received much less attention.

There are other avenues to enhance expressiveness. The idea of *abstract syntax graphs* (ASGs) [Oliveira and Cook, 2012b; Oliveira and Löh, 2013] encodes sharing and loops in data structures. Potentially we could generate ASGs with generic unfolds in applications like parsing and enumeration, to optimize data representations and execution efficiency.

More Exploration in Theory We believe that more exploration can be done from a theoretical or mathematical perspective, to answer the following questions:

- What is the relationship between the difference of internal visitors versus external visitors, and unfold-based versus fold-based transformations (Section 5.4.4)? Where to use the fold-based transformations?
- Besides consumers and producers, can we identify transformations as an independent recursion pattern?
- Chapter 5 showed that while algebra composition leads to the sum of functors, coalgebra composition refers to the product type, and a subsequent transformation changes it to the sum type. Can we express the nice duality between algebras and coalgebras as in category theory [Herrlich and Strecker, 1973]? What is the connection to the duality between Church encoding and co-Church encoding?

Long-Term Goals There are two long-term goals for the area of modularization. Firstly, we would like to introduce generic producers and their composition in object-oriented programming, similar to Object Algebras which realize generic consumers (traversals) and extensibility in OOP. Secondly, existing languages are not powerful enough to represent consumers, producers and their extensibility. Due to the lack of advanced language features, our techniques have to encode those patterns sometimes with cumbersome boilerplate to ensure type safety. As argued in Section 1.2, modularization techniques should spur the evolution of programming languages; we expect a language/calculus to modularize data structures and various operations in the first place, and involve deforestation techniques to optimize the performance. At this point, this dissertation is hopefully a stepping stone for further exploration.

Bibliography

Michael D. Adams and Thomas M. DuBuisson. 2012. Template Your Boilerplate: Using Template Haskell for Efficient Generic Programming. In *Proceedings of the 2012 ACM SIGPLAN Haskell symposium (Haskell '12)*. 13–24. [cited on pages: 107, 135, and 142]

Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *Journal of Object Technology* 8, 5 (2009), 49–84. [cited on page: 4]

John W. Backus. 1954. Specifications for the IBM mathematical FORmula TRANslation system. *New York: IBM Applied Science Division* 10 (1954). [cited on page: 1]

Patrick Bahr and Tom Hvitved. 2011. Compositional data types. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming (WGP '11)*. ACM, New York, NY, USA, 83–94. [cited on pages: 130, 131, and 132]

Kent Beck and Erich Gamma. 1998. Test infected: Programmers love writing tests. *Java Report* 3, 7 (1998), 37–50. [cited on page: 26]

Eric Béguet and Manohar Jonnalagedda. 2014. Accelerating Parser Combinators with Macros. In *Proceedings of SCALA 2014*. 7–17. [cited on page: 75]

Aggelos Biboudis, Pablo Inostroza, and Tijs van der Storm. 2016. Recaf: Java Dialects As Libraries. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. 2–13. [cited on page: 138]

Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. 2005. The Essence of Data Access in Comega. In *ECOOP 2005*. [cited on page: 135]

Richard Bird and Oege de Moor. 1997. *Algebra of Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [cited on pages: 5, 8, 13, 24, 77, 129, 130, and 132]

Corrado Böhm and Alessandro Berarducci. 1985. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science* 39 (1985), 135 – 154. Third Conference on Foundations of Software Technology and Theoretical Computer Science. [cited on page: 130]

Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (1966), 366–371. [cited on page: 2]

Peter Borovanskỳ, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. 1996. ELAN: A logical framework based on computational systems. *Electronic Notes in Theoretical Computer Science* 4 (1996), 35–50. [cited on pages: 6, 133, and 143]

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)*. 303–311. [cited on page: 62]

Martin Bravenboer and Eelco Visser. 2001. *Guiding visitors: Separating navigation from computation*. Technical Report UU-CS-2001-42. Institute of Information and Computing Sciences, Utrecht University. [cited on page: 135]

Martin Bravenboer and Eelco Visser. 2008. Parse Table Composition. In *Proceedings of SLE 2008*. 74–94. [cited on page: 137]

Peter Buchlovsky and Hayo Thielecke. 2006. A Type-theoretic Reconstruction of the Visitor Pattern. *Electron. Notes Theor. Comput. Sci.* 155 (May 2006), 309–329. [cited on pages: 18, 21, 130, 131, and 132]

William H. Burge. 1975. *Recursive programming techniques*. Addison-Wesley Longman, Incorporated. [cited on pages: 27, 61, and 137]

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989. F-bounded Polymorphism for Object-oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. 273–280. [cited on pages: 5, 21, and 130]

Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. 2007. Finally Tagless, Partially Evaluated: Tagless Staged Interpreters for Simpler Typed Languages. In *Proceedings of the 5th Asian Conference on Programming Languages and Systems (APLAS '07)*. 222–238. [cited on pages: 3, 23, and 130]

Walter Cazzola and Edoardo Vacchi. 2016. Language Components for Modular DSLs Using Traits. *Comput. Lang. Syst. Struct.* 45, C (April 2016), 16–34. [cited on page: 131]

Bryan Chadwick and Karl Lieberherr. 2010. Weaving Generic Programming and Traversal Performance. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10)*. 61–72. [cited on page: 133]

Craig Chambers and Gary T. Leavens. 1995. Typechecking and Modules for Multimethods. *ACM Trans. Program. Lang. Syst.* 17, 6 (Nov. 1995), 805–843. [cited on page: 21]

Jan Christiansen and Sebastian Fischer. 2008. EasyCheck — Test Data for Free. In *Functional and Logic Programming*, Jacques Garrigue and Manuel V. Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 322–336. [cited on page: 139]

Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. [cited on page: 2]

Alonzo Church. 1936. An Unsolvable Problem of Elementary Number Theory. *Journal of Symbolic Logic* 1, 2 (1936), 73–74. [cited on pages: 18 and 130]

Koen Claessen, Jonas Duregård, and Michał H. Pałka. 2014. Generating Constrained Random Data with Uniform Distribution. In *Functional and Logic Programming*. Springer International Publishing, Cham, 18–34. [cited on pages: 78, 79, 122, and 139]

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*. 268–279. [cited on pages: 30, 78, 103, and 139]

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. 130–145. [cited on page: 21]

William R. Cook. 1989. *A Denotational Semantics of Inheritance*. Ph.D. Dissertation. Brown University. [cited on page: 72]

Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)* 75, 5 (1972), 381 – 392. [cited on pages: 48 and 118]

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à La Carte. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. 207–218. [cited on page: 138]

Edsger W. Dijkstra. 1968. Letters to the Editor: Go to Statement Considered Harmful. *Commun. ACM* 11, 3 (March 1968), 147–148. [cited on pages: 1 and 2]

Robert Dondero. 2014. `https://www.cs.princeton.edu/courses/archive/fall14/cos217/`. Accessed: 2019-01-17. [cited on page: 2]

Jonas Duregård. 2012. *Enumerative Testing and Embedded Languages*. Ph.D. Dissertation. Chalmers University of Technology. [cited on page: 124]

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: Functional Enumeration of Algebraic Types. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. ACM, New York, NY, USA, 61–72. [cited on pages: 78, 79, and 139]

Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of OOPSLA 2007*. 1–18. [cited on pages: 80, 137, and 138]

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: library-based syntactic language extensibility. In *Proceedings of OOPSLA 2011*. 391–406. [cited on pages: 136 and 138]

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 *Software Language Engineering*. Springer. [cited on pages: 34 and 54]

Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24–47. [cited on pages: 4 and 143]

Erik Ernst. 2001. Family Polymorphism. In *Proceedings of ECOOP 2001*. 303–326. [cited on pages: 4, 72, and 142]

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *Proceedings of POPL 2006*. 270–282. [cited on page: 4]

Martin Odersky et al. 2004. *An Overview of the Scala Programming Language*. Technical Report IC/2004/64. EPFL Lausanne, Switzerland. [cited on pages: 61 and 66]

Burke Fetscher, Koen Claessen, Michał Pałka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *ESOP 2015*. 383–405. [cited on pages: 78, 79, 122, and 139]

Maarten Fokkinga. 1994. *Monadic Maps and Folds for Arbitrary Datatypes*. Technical Report. Memoranda Informatica, University of Twente. [cited on pages: 5 and 16]

Bryan Ford. 2002. Packrat parsing: : simple, powerful, lazy, linear time, functional pearl. In *Proceedings of ICFP 2002*. 36–47. [cited on pages: 9, 27, 61, 62, 64, 131, and 137]

Richard A. Frost, Rahmatullah Hafiz, and Paul Callaghan. 2008. Parser Combinators for Ambiguous Left-Recursive Grammars. In *Proceedings of PADL 2008*. 167–181. [cited on page: 137]

Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software*. Pearson Education India. [cited on pages: 2, 16, 18, 20, 129, 134, and 135]

Jeremy Gibbons. 2003a. Origami Programming. In *The Fun of Programming*. Palgrave, 41–60. [cited on page: 129]

Jeremy Gibbons. 2003b. Patterns in Datatype-Generic Programming. In *Multiparadigm Programming*, Vol. 27. John von Neumann Institute for Computing (NIC), 277–289. First International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DP-COOL). [cited on page: 129]

Jeremy Gibbons. 2006. Design Patterns As Higher-order Datatype-generic Programs. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming (WGP '06)*. 1–12. [cited on page: 129]

Jeremy Gibbons and Bruno C. d. S. Oliveira. 2009. The essence of the Iterator pattern. *Journal of Functional Programming* 19, 3-4 (2009), 377–402. [cited on pages: 16 and 129]

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. [cited on page: 24]

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. 223–232. [cited on pages: 5 and 132]

Maria Gouseti, Chiel Peters, and Tijs van der Storm. 2014. Extensible Language Implementation with Object Algebras (Short Paper). In *GPCE'14*. [cited on pages: 4, 8, 34, 54, 61, and 136]

Robert Grimm. 2006. Better extensibility through modular syntax. In *Proceedings of PLDI 2006*. 38–51. [cited on pages: 4, 8, 61, and 136]

Katarzyna Grygiel and Pierre Lescanne. 2013. Counting and generating lambda terms. *Journal of Functional Programming* 23, 5 (2013), 594–628. [cited on pages: 78, 79, and 139]

Jan Hannemann and Gregor Kiczales. 2002. Design Pattern Implementation in Java and AspectJ. In *OOPSLA '02*. [cited on page: 136]

Jan Heering, Paul Robert Hendrik Hendriks, Paul Klint, and Jan Rekers. 1989. The syntax definition formalism SDF–reference manual–. *ACM Sigplan Notices* 24, 11 (1989), 43–75. [cited on page: 137]

Horst Herrlich and George Strecker. 1973. Category theory. (1973). [cited on pages: 5, 8, 13, 129, 132, and 144]

Ralf Hinze. 2006. Generics for the Masses. *J. Funct. Program.* 16, 4-5 (July 2006), 451–483. [cited on pages: 23 and 130]

Ralf Hinze. 2010. A category theory primer. `http://www.cs.ox.ac.uk/ralf.hinze/SSGIP10/Notes.pdf` SSGIP '10 Notes. [cited on pages: 91 and 133]

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Proceedings of the 22Nd International Conference on Implementation and Application of Functional Languages (IFL '10)*. 19–37. [cited on pages: 90, 132, and 133]

Furio Honsell, Marina Lenisa, and Rekha Redamalla. 2004. Coalgebraic Semantics and Observational Equivalences of an Imperative Class-based OO-Language. *Electron. Notes Theor. Comput. Sci.* 104 (Nov. 2004), 163–180. [cited on page: 138]

John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (April 1989), 98–107. [cited on page: 2]

Graham Hutton. 1998. Fold and Unfold for Program Semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. 280–288. [cited on pages: 110 and 138]

Graham Hutton and Erik Meijer. 1996. *Monadic parser combinators*. Technical Report NOTTCS-TR-96-4. University of Nottingham. `http://eprints.nottingham.ac.uk/237/` [cited on page: 137]

Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *J. Funct. Program.* 8, 4 (July 1998), 437–444. [cited on page: 113]

Pablo Inostroza and Tijs van der Storm. 2015. Modular Interpreters for the Masses: Implicit Context Propagation Using Object Algebras. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '15)*. 171–180. [cited on pages: 80 and 138]

Mauro Jaskelioff, Neil Ghani, and Graham Hutton. 2011. Modularity and Implementation of Mathematical Operational Semantics. *Electronic Notes in Theoretical Computer Science* 229, 5 (2011), 75 – 95. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). [cited on page: 138]

Patricia Johann and Neil Ghani. 2008. Foundations for Structured Programming with GADTs. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. 297–308. [cited on pages: 130 and 144]

Christian Kästner, Sven Apel, and Klaus Ostermann. 2011. The road to feature modularity?. In *Proceedings of SPLC 2011*. 5:1–5:8. [cited on page: 4]

Oleg Kiselyov. 2012. Typed Tagless Final Interpreters. In *Proceedings of the 2010 International Spring School Conference on Generic and Indexed Programming (SSGIP '10)*. 130–174. [cited on page: 130]

Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE Computer Society, Washington, DC, USA, 168–177. [cited on page: 138]

Donald E. Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (01 Jun 1968), 127–145. [cited on page: 131]

Ralf Lammel and Simon Peyton Jones. 2003. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming. In *TLDI '03*. [cited on pages: 5, 6, 8, 40, and 135]

Ralf Lämmel and Simon Peyton Jones. 2004. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts. In *ICFP '04*. [cited on page: 135]

Ralf Lämmel and Simon Peyton Jones. 2005. Scrap Your Boilerplate with Class: Extensible Generic Functions. In *ICFP '05*. [cited on page: 135]

Ralf Lämmel, Eelco Visser, and Joost Visser. 2003. Strategic programming meets adaptive programming. In *AOSD '03*. [cited on page: 133]

Ralf Lämmel and Joost Visser. 2002. Typed combinators for generic traversal. In *Practical Aspects of Declarative Languages*. Springer, 137–154. [cited on page: 133]

Ralf Lämmel, Joost Visser, and Jan Kort. 2000. Dealing with Large Bananas. In *Workshop on Generic Programming*, Johan Jeuring (Ed.). Technical Report UU-CS-2000-19, Universiteit Utrecht, Ponte de Lima. [cited on page: 135]

M. Leduc, T. Degueule, B. Combemale, T. van der Storm, and O. 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)*, Vol. 00. 112–122. [cited on page: 138]

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. Technical Report UU-CS-2001-3. Department of Information and Computing Sciences, Utrecht University. [cited on pages: 27, 61, 63, 113, and 137]

Marina Lenisa, John Power, and Hiroshi Watanabe. 2004. Category Theory for Operational Semantics. *Theor. Comput. Sci.* 327, 1-2 (Oct. 2004), 135–154. [cited on page: 138]

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. 333–343. [cited on pages: 28, 80, 83, 100, 110, and 139]

K. J. Lieberherr. 1996. *Adaptive Object Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing. [cited on pages: 6 and 133]

Barbara Liskov and Stephen Zilles. 1974. Programming with Abstract Data Types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. 50–59. [cited on page: 2]

Conor Mcbride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. [cited on page: 16]

Lambert Meertens. 1992. Paramorphisms. *Formal Aspects of Computing* 4, 5 (1992), 413–424. [cited on pages: 5 and 15]

Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*. 124–144. [cited on page: 131]

Nax Paul Mendler. 1991. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic* 51, 1 (1991), 159 – 172. [cited on pages: 116, 137, and 143]

Matthew Might, David Darais, and Daniel Spiewak. 2011. Parsing with derivatives: a functional pearl. In *Proceeding of ICFP 2011*. 189–195. [cited on page: 137]

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. *Parser combinators in Scala*. Technical Report. Department of Computer Science, K.U. Leuven. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html` [cited on page: 64]

Martin E Nordberg III. 1996. Variations on the visitor pattern. In *PLoP '96 Writer's Workshop*, Vol. 154. [cited on pages: 6 and 134]

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of CC 2003*. 138–152. [cited on pages: 80, 137, and 138]

Martin Odersky and Matthias Zenger. 2005a. Independently extensible solutions to the expression problem. In *FOOL '05*, Vol. 12. [cited on pages: 3, 5, 21, 66, 72, and 130]

Martin Odersky and Matthias Zenger. 2005b. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. 41–57. [cited on page: 21]

Bruno C. d. S. Oliveira. 2007. *Genericity, extensibility and type-safety in the VISITOR pattern*. Ph.D. Dissertation. University of Oxford, UK. [cited on pages: 18 and 129]

Bruno C. d. S. Oliveira. 2009. Modular Visitor Components. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. 269–293. [cited on page: 138]

Bruno C. d. S. Oliveira and William R. Cook. 2012a. Extensibility for the Masses, Practical Extensibility with Object Algebras. In *ECOOP '12*. [cited on pages: 3, 5, 8, 9, 18, 20, 21, 33, 34, 67, 130, 135, 138, 143, and 144]

Bruno C. d. S. Oliveira and William R. Cook. 2012b. Functional Programming with Structured Graphs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 77–88. [cited on page: 144]

Bruno C. d. S. Oliveira, Ralf Hinze, and Andres Löh. 2006. Extensible and modular generics for the masses. *Trends in Functional Programming* 7 (2006), 199–216. [cited on pages: 23 and 130]

Bruno C. d. S. Oliveira and Andres Löh. 2013. Abstract Syntax Graphs for Domain Specific Languages. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM '13)*. 87–96. [cited on page: 144]

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)*. 82–93. [cited on pages: 91, 128, 130, 131, 132, and 143]

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 (ECOOP '13)*. 27–51. [cited on pages: 5, 18, 131, and 138]

Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. 2008. The Visitor Pattern As a Reusable, Generic, Type-safe Component. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA '08)*. 439–456. [cited on pages: 18, 129, 131, and 135]

Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2014. Safely Composable Type-Specific Languages. In *Proceedings of ECOOP 2014*. 105–130. [cited on page: 136]

Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. 1997. A Calculational Fusion System HYLO. In *Proceedings of the IFIP TC 2 WG 2.1 International Workshop on Algorithmic Languages and Calculi*. 76–106. [cited on page: 132]

Doug Orleans and Karl Lieberherr. 2001. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001*. Springer-Verlag. [cited on page: 133]

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST '11)*. 91–97. [cited on pages: 78, 79, and 139]

Jens Palsberg and C. Barry Jay. 1998. The Essence of the Visitor Pattern. In *COMPSAC '98*. [cited on page: 134]

Alberto Pardo. 1998. Monadic Corecursion -Definition, Fusion Laws, and Applications-. *Electronic Notes in Theoretical Computer Science* 11 (1998), 105 – 139. CMCS '98, First Workshop on Coalgebraic Methods in Computer Science. [cited on pages: 5, 16, 91, 94, 96, 116, 132, and 137]

Michel Parigot. 1992. Recursive Programming with Proofs. *Theor. Comput. Sci.* 94, 2 (March 1992), 335–356. [cited on page: 131]

David Lorge Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. [cited on page: 2]

Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL(K) Parser Generator. *Softw. Pract. Exper.* 25, 7 (July 1995), 789–810. [cited on pages: 4, 8, 61, and 136]

Benjamin C. Pierce. 2002. *Types and programming languages*. MIT press. [cited on pages: 8, 62, 72, 113, and 117]

Gordon D. Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19. University of Aarhus. `http://citeseer.ist.psu.edu/plotkin81structural.html` [cited on page: 138]

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 & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 377–395. [cited on page: 131]

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell (Haskell '08)*. 37–48. [cited on page: 139]

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behaviour. In *ECOOP '03*. Springer Berlin Heidelberg, Berlin, Heidelberg, 248–274. [cited on page: 62]

Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. 2009. Complete and Decidable Type Inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*. ACM, New York, NY, USA, 341–352. [cited on page: 130]

August Schwerdfeger and Eric Van Wyk. 2009a. Verifiable Composition of Deterministic Grammars. In *Proceedings of PLDI 2009*. 199–210. [cited on pages: 4, 8, 61, and 136]

August Schwerdfeger and Eric Van Wyk. 2009b. Verifiable Parse Table Composition for Deterministic Parsing. In *Proceedings of SLE 2009*. 184–203. [cited on page: 137]

Dana S. Scott and Christopher Strachey. 1971. *Toward a mathematical semantics for computer languages*. Vol. 1. Oxford University Computing Laboratory, Programming Research Group. [cited on page: 138]

Anthony M. Sloane and Matthew Roberts. 2015. Oberon-0 in Kiama. *Science of Computer Programming* 114 (2015), 20 – 32. [cited on page: 138]

William Steingartner, Valerie Novitzká, Mohamed Ali M. Eldojali, and Davorka Radaković. 2016. Some aspects about coalgebras as foundation for expressing the semantics of imperative languages. Mathematics and Computer Science-MaCS 2016. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science. Eger, Hungary*. [cited on page: 138]

Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436. [cited on pages: 3, 8, 13, 24, 72, 77, 81, 86, 113, 120, 129, 130, 131, 132, and 138]

Akihiko Takano and Erik Meijer. 1995. Shortcut Deforestation in Calculational Form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. 306–313. [cited on page: 132]

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *Proceedings of PLDI 2011*. 132–141. [cited on pages: 136 and 137]

Mads Torgersen. 2004. The Expression Problem Revisited. In *ECOOP '04*. 123–146. [cited on pages: 3, 5, 21, and 130]

Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. 2004. Adding Wildcards to the Java Programming Language. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*. 1289–1296. [cited on page: 130]

Daniele Turi and Gordon D. Plotkin. 1997. Towards a Mathematical Operational Semantics. In *LICS*. [cited on page: 138]

Tarmo Uustalu and Varmo Vene. 2000. Coding recursion a la Mendler. In *WGP '00*. 69–85. [cited on pages: 116, 137, 138, and 143]

Mark G. J. van den Brand, Paul Klint, and Jurgen J. Vinju. 2003a. Term Rewriting with Traversal Functions. *ACM Trans. Softw. Eng. Methodol.* 12, 2 (April 2003), 152–190. [cited on page: 134]

Mark G. J. van den Brand, Paul Klint, and Jurgen J. Vinju. 2003b. Term Rewriting with Traversal Functions. *ACM Trans. Softw. Eng. Methodol.* 12, 2 (April 2003), 152–190. [cited on page: 143]

Varmo Vene. 2000. *Categorical programming with inductive and coinductive types*. Ph.D. Dissertation. University of Tartu. [cited on page: 132]

Varmo Vene and Tarmo Uustalu. 1998. Functional programming with apomorphisms (corecursion). In *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, Vol. 47. 147–161. [cited on pages: 5, 15, and 111]

Marcos Viera, Doaitse Swierstra, and Atze Dijkstra. 2012. Grammar Fragments Fly First-class. In *Proceedings of LDTA 2012*. 5:1–5:7. [cited on pages: 4, 8, 61, and 136]

Eelco Visser. 2001a. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of RTA 2001*. 357–362. [cited on page: 137]

Eelco Visser and Zine-el-Abidine Benaissa. 1998a. A core language for rewriting. *Electronic Notes in Theoretical Computer Science* 15 (1998), 422–441. [cited on pages: 6, 133, and 134]

Eelco Visser and Zine-el-Abidine Benaissa. 1998b. A core language for rewriting. *Electronic Notes in Theoretical Computer Science* 15 (1998), 422–441. [cited on page: 143]

Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. 1998. Building Program Optimizers with Rewriting Strategies. In *ICFP'98*. [cited on pages: 133 and 134]

Joost Visser. 2001b. Visitor combination and traversal control. In *OOPSLA '01*. [cited on pages: 133 and 134]

Philip Wadler. 1985. How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Proceedings of Functional Programming Languages and Computer Architecture, 1985*. 113–128. [cited on pages: 61 and 137]

Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *Proceedings of the 2Nd European Symposium on Programming (ESOP '88)*. 344–358. [cited on pages: 5, 91, and 132]

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. 61–78. [cited on page: 98]

Philip Wadler. 1998. The Expression Problem. Email. Discussion on the Java Genericity mailing list. [cited on pages: 2, 16, 61, 67, 129, and 141]

Philip Wadler and Stephen Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. 60–76. [cited on page: 135]

Jue Wang. 2005. *Generating Random Lambda Calculus Terms*. Technical report. Boston University. [cited on page: 139]

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. [cited on pages: 3, 18, 21, 67, and 130]

Alessandro Warth, James R. Douglass, and Todd D. Millstein. 2008. Packrat parsers can support left recursion. In *Proceedings of PEPM 2008*. 103–110. [cited on page: 64]

Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. 2016a. Modular semantic actions. In *Proceedings of DLS 2016*. 108–119. [cited on pages: 4, 8, and 136]

Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. 2016b. Modular Semantic Actions. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 108–119. [cited on pages: 61 and 138]

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 (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 74. 29:1–29:32. [cited on pages: 5, 117, 131, and 138]

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 (GPCE 2018)*. 134–146. [cited on page: 117]

Appendix A

Complete Code for Chapter 3

A.1 OO Approach for `usedVars` and `rename`

Below is the complete code for Figure 3.2. It implements `usedVars` and `rename` in the QL example, as an OO approach.

```
class Form {
   String name;
   List<Stmt> body;
   Form(String id, List<Stmt> body) {
      this.name = id;
      this.body = new ArrayList<Stmt>(body);
   }
   Set<String> usedVars() {
      Set<String> vars = new HashSet<>();
      body.forEach(s -> vars.addAll(s.usedVars()));
      return vars;
   }
   Form rename(String n1, String n2) {
      List<Stmt> ss = new ArrayList<>();
      for (Stmt s: body) ss.add(s.rename(n1, n2));
      return new Form(name, ss);
   }
}

abstract class Stmt {
   abstract Set<String> usedVars();
   abstract Stmt rename(String n1, String n2);
}

class If extends Stmt {
   Exp cond;
   Stmt then;
```

```
   If(Exp cond, Stmt then) {
      this.cond = cond;
      this.then = then;
   }
   Set<String> usedVars() {
      Set<String> vars = new HashSet<>(cond.usedVars());
      vars.addAll(then.usedVars());
      return vars;
   }
   If rename(String n1, String n2) {
      return new If(cond.rename(n1, n2), then.rename(n1, n2));
   }
}

class Question extends Stmt {
   String name, label, type;
   Question(String n, String l, String t) {
      this.name = n;
      this.label = l;
      this.type = t;
   }
   Set<String> usedVars() {
      return emptySet();
   }
   Question rename(String n1, String n2) {
      String newN = name.equals(n1) ? n2 : name;
      return new Question(newN, label, type);
   }
}


abstract class Exp {
   abstract Set<String> usedVars();
   abstract Exp rename(String n1, String n2);
}

class Lit extends Exp {
   int n;
   Lit(int n) {
      this.n = n;
   }
   Set<String> usedVars() {
      return emptySet();
   }
   Lit rename(String n1, String n2) {
```

```
      return new Lit(n);
   }
}

class Var extends Exp {
   String x;
   Var(String name) {
      this.x = name;
   }
   Set<String> usedVars() {
      return Collections.singleton(x);
   }
   Var rename(String n1, String n2) {
      String newN = x.equals(n1) ? n2 : x;
      return new Var(newN);
   }
}

class GEq extends Exp {
   Exp lhs, rhs;
   GEq(Exp lhs, Exp rhs) {
      this.lhs = lhs;
      this.rhs = rhs;
   }
   Set<String> usedVars() {
      Set<String> vars = new HashSet<>(lhs.usedVars());
      vars.addAll(rhs.usedVars());
      return vars;
   }
   GEq rename(String n1, String n2) {
      return new GEq(lhs.rename(n1, n2), rhs.rename(n1, n2));
   }
}
```

A.2 `Rename` implementing the `QLAlg` interface

The following code gives the implementation of `Rename` that implements `QLAlg` in Section 3.1.2.

```
class Rename<E, S, F> implements QLAlg<E, S, F> {
   private QLAlg<E, S, F> alg;
   private String from, to;
   public Rename(QLAlg<E, S, F> alg, String from, String to) {
      this.alg = alg;
      this.from = from;
      this.to = to;
   }
   public F Form(String id, List<S> stmts) {
      return alg.Form(id, stmts);
   }
   public S If(E c, S t) {
      return alg.If(c, t);
   }
   public S Question(String n, String l, String t) {
      n = n.equals(from) ? to : n;
      return alg.Question(n, l, t);
   }
   public E Lit(int n) {
      return alg.Lit(n);
   }
   public E Var(String x) {
      x = x.equals(from) ? to : x;
      return alg.Var(x);
   }
   public E GEq(E l, E r) {
      return alg.GEq(l, r);
   }
}
```

A.3 `QLAlgQuery`: generated code

The generated code for `QLAlgQuery` by *Shy* in Figure 3.5.

```
public interface QLAlgQuery<R> extends QLAlg<R, R, R> {

   Monoid<R> m();

   default R Form(java.lang.String p0, java.util.List<R> p1) {
      R res = m().empty();
      res = m().join(res, m().fold(p1));
      return res;
```

```
    }

    default R Geq(R p0, R p1) {
       R res = m().empty();
       res = m().join(res, p0);
       res = m().join(res, p1);
       return res;
    }

    default R If(R p0, R p1) {
       R res = m().empty();
       res = m().join(res, p0);
       res = m().join(res, p1);
       return res;
    }

    default R Lit(int p0) {
       R res = m().empty();
       return res;
    }

    default R Question(java.lang.String p0, java.lang.String p1,
          java.lang.String p2) {
       R res = m().empty();
       return res;
    }

    default R Var(java.lang.String p0) {
       R res = m().empty();
       return res;
    }

}
```

A.4 `QLAlgTransform` and `QLAlgTrans`: generated code

The code for `QLAlgTransform` and its class representation `QLAlgTrans` for use, generated by *Shy*. See Figure 3.6.

```
public interface QLAlgTransform<A0, A1, A2> extends QLAlg<A0, A1, A2> {

   QLAlg<A0, A1, A2> qLAlg();

   default A2 Form(java.lang.String p0, java.util.List<A1> p1) {
      return qLAlg().Form(p0, p1);
   }

   default A0 Geq(A0 p0, A0 p1) {
      return qLAlg().Geq(p0, p1);
   }

   default A1 If(A0 p0, A1 p1) {
      return qLAlg().If(p0, p1);
   }

   default A0 Lit(int p0) {
      return qLAlg().Lit(p0);
   }

   default A1 Question(java.lang.String p0, java.lang.String p1,
         java.lang.String p2) {
      return qLAlg().Question(p0, p1, p2);
   }

   default A0 Var(java.lang.String p0) {
      return qLAlg().Var(p0);
   }

}

public class QLAlgTrans<A0, A1, A2> implements QLAlgTransform<A0, A1, A2> {

   private QLAlg<A0, A1, A2> alg;

   public QLAlgTrans(QLAlg<A0, A1, A2> alg) {
      this.alg = alg;
   }

   public QLAlg<A0, A1, A2> qLAlg() {return alg;}
```

```
}
```

A.5 G_ExpAlgQuery: generated code

The generated code for G_ExpAlgQuery by *Shy* in Figure 3.10.

```
public interface G_ExpAlgQuery<A0> extends ExpAlg<A0> {

   Monoid<A0> mExp();

   default A0 Add(A0 p0, A0 p1) {
      A0 res = mExp().empty();
      res = mExp().join(res, p0);
      res = mExp().join(res, p1);
      return res;
   }

   default A0 Lit(int p0) {
      A0 res = mExp().empty();
      return res;
   }

   default A0 Var(java.lang.String p0) {
      A0 res = mExp().empty();
      return res;
   }

}
```

Appendix B

Complete Code for Chapter 5

B.1 `weightedTrafo`: the natural transformation for weighted distribution

The following code gives the natural transformation for weighted distribution, which was used to derive $|*|_w$ in Section 5.5.2.

```
weightedTrafo :: Cardinality g => Trafo Weighted f g
weightedTrafo (Prod (Comp fs) (Comp gs)) = Comp . MaybeT $ do
  p  <- pointer
  rF <- runMaybeT $ fmap Inl fs
  rG <- runMaybeT $ succ >> fmap Inr gs
  when (card (getProxy gs) == 1 && isJust rG) (getWeight (p + 1) >>= setAcc)
  wF <- getWeight p
  wG <- getAcc
  case (rF, rG) of
    (Just _, Just _) -> setAcc (wF + wG) >> binomial (wF, return rF)
                                                     (wG, return rG)
    (Just _, _     ) -> setAcc wF >> return rF
    (_     , Just _) -> return rG
    _                -> return Nothing
  where getProxy :: m (h a) -> Proxy h
        getProxy _ = Proxy
```

B.2 `weight`: the weight function in dynamic distribution

The following code gives the implementation of the weight function used in Figure 5.21.

```
weight :: (Int, Int) -> (Int, Int) -> Double
weight (maxArity, maxDepth) (thisArity, thisDepth) =
  prob thisArity maxArity (fromIntegral (1 + thisDepth) / fromIntegral (2 +
    maxDepth))
```

```
prob :: Int -> Int -> Double -> Double
prob k n p = fromIntegral (choose n k) * pow p k * pow (1 - p) (n - k)
  where choose :: Int -> Int -> Int
        choose n 0 = 1
        choose 0 k = 0
        choose n k = choose (n - 1) (k - 1) * n `div` k
        pow :: Double -> Int -> Double
        pow x 0 = 1
        pow x n = x * pow x (n - 1)
```

B.3 `IsNumericVal`, `rdcRule`, `cgrRule`,

The following code defines the `IsNumericVal` class to check if an expression has the form of a numeric value, used in Figure 5.24.

```
class Functor f => IsNumericVal (f :: * -> *) where
  isNumAlg :: Alg f Bool

isNum :: IsNumericVal f => Fix f -> Bool
isNum = fold isNumAlg

instance IsNumericVal ArithF where
  isNumAlg TmZero        = True
  isNumAlg (TmSucc True) = True
  isNumAlg _             = False

instance IsNumericVal BoolF where
  isNumAlg _ = False

instance (IsNumericVal f, IsNumericVal g) => IsNumericVal (f ⊕ g) where
  isNumAlg (Inl x) = isNumAlg x
  isNumAlg (Inr x) = isNumAlg x

rdcRule :: MonadPrior IsRdc Conflict m => a -> m a
rdcRule = create True

cgrRule :: MonadPrior IsRdc Conflict m => a -> m a
cgrRule = create False
```

B.4 Smart constructors for `ArithF` and `BoolF`

The following code defines the smart constructors for `ArithF` and `BoolF` in Section 5.5.3.

```
zero :: ArithF :<: f => Fix f
zero = In . inj $ TmZero
```

```
succ :: ArithF :<: f => Fix f -> Fix f
succ = In . inj . TmSucc

pred :: ArithF :<: f => Fix f -> Fix f
pred = In . inj . TmPred

iszero :: ArithF :<: f => Fix f -> Fix f
iszero = In . inj . TmIsZero

true :: BoolF :<: f => Fix f
true = In . inj $ TmTrue

false :: BoolF :<: f => Fix f
false = In . inj $ TmFalse

ifC :: BoolF :<: f => Fix f -> Fix f -> Fix f -> Fix f
ifC x y z = In . inj $ TmIf x y z
```

Appendix C

Complete Code for Chapter 6

C.1 Projection

Some projection pattterns for `LNG` are defined below.

```
pattern LitP x     <- (proj . out -> Just (Lit x))
pattern AddP x y   <- (proj . out -> Just (Add x y))
pattern MulP x y   <- (proj . out -> Just (Mul x y))
pattern BoolP x    <- (proj . out -> Just (BoolV x))
pattern IfP x y z  <- (proj . out -> Just (If x y z))
pattern EqualP x y <- (proj . out -> Just (Equal x y))
pattern VarP x     <- (proj . out -> Just (Var x))
pattern LamP t x   <- (proj . out -> Just (Lam t x))
pattern AppP x y   <- (proj . out -> Just (App x y))
```

C.2 Type-checker

Below is the code for type-checking `Fix LNG` expressions.

```
type TCEnv = [Type]

tcheckAdd :: TCEnv -> Fix LNG -> Fix LNG -> Maybe Type
tcheckAdd env x y = do
  TLit <- tcheck env x
  TLit <- tcheck env y
  return TLit

tcheckMul :: TCEnv -> Fix LNG -> Fix LNG -> Maybe Type
tcheckMul env x y = do
  TLit <- tcheck env x
  TLit <- tcheck env y
  return TLit
```

```
tcheckIf :: TCEnv -> Fix LNG -> Fix LNG -> Fix LNG -> Maybe Type
tcheckIf env x y z = do
  TBool <- tcheck env x
  t1 <- tcheck env y
  t2 <- tcheck env z
  guard $ t1 == t2
  return t1

tcheckEqual :: TCEnv -> Fix LNG -> Fix LNG -> Maybe Type
tcheckEqual env x y = do
  t1 <- tcheck env x
  t2 <- tcheck env y
  guard $ t1 == t2
  guard $ t1 == TBool || t1 == TLit
  return TBool

tcheckVar :: TCEnv -> Int -> Maybe Type
tcheckVar env x
  | x >= 0 && x < length env = Just $ env !! x
  | otherwise = Nothing

tcheckLam :: TCEnv -> Type -> Fix LNG -> Maybe Type
tcheckLam env t e = do
  t1 <- tcheck (t:env) e
  return $ TFunc t t1

tcheckApp :: TCEnv -> Fix LNG -> Fix LNG -> Maybe Type
tcheckApp env x y = do
  t1 <- tcheck env x
  t2 <- tcheck env y
  case t1 of
    TFunc t0 t -> if t0 == t2 then return t else Nothing
    _ -> Nothing

tcheck :: [Type] -> Fix LNG -> Maybe Type
tcheck env (LitP _)     = Just TLit
tcheck env (AddP x y)   = tcheckAdd env x y
tcheck env (MulP x y)   = tcheckMul env x y
tcheck env (BoolP _)    = Just TBool
tcheck env (IfP x y z)  = tcheckIf env x y z
tcheck env (EqualP x y) = tcheckEqual env x y
tcheck env (VarP x)     = tcheckVar env x
tcheck env (LamP t x)   = tcheckLam env t x
tcheck env (AppP x y)   = tcheckApp env x y
```

```
typeCheck :: Fix LNG -> Maybe Type
typeCheck = tcheck []
```

C.3 Evaluation

Below is the semantic evaluation code for `Fix LNG`.

```
projLit :: LitF :<: f => Fix f -> Maybe Int
projLit (In e) = case proj e of
  Just (Lit x) -> Just x
  _            -> Nothing

projBool :: BoolF :<: f => Fix f -> Maybe Bool
projBool (In e) = case proj e of
  Just (BoolV x) -> Just x
  _              -> Nothing

projLam :: LamF :<: f => Fix f -> Maybe (Type, Fix f)
projLam (In e) = case proj e of
  Just (Lam t e') -> Just (t, e')
  _               -> Nothing

evalAdd :: Fix LNG -> Fix LNG -> Maybe (Fix LNG)
evalAdd (LitP x) (LitP y) = return . lit $ x + y
evalAdd e1       e2       = case (eval e1, eval e2) of
    (Just e1', _) -> return $ add e1' e2
    (_, Just e2') -> return $ add e1 e2'
    _             -> Nothing

evalMul :: Fix LNG -> Fix LNG -> Maybe (Fix LNG)
evalMul (LitP x) (LitP y) = return . lit $ x * y
evalMul e1       e2       = case (eval e1, eval e2) of
    (Just e1', _) -> return $ mul e1' e2
    (_, Just e2') -> return $ mul e1 e2'
    _             -> Nothing

evalIf :: Fix LNG -> Fix LNG -> Fix LNG -> Maybe (Fix LNG)
evalIf (BoolP True)  e  _  = Just e
evalIf (BoolP False) _  e  = Just e
evalIf e1            e2 e3 = case (eval e1, eval e2, eval e3) of
    (Just e1', _, _) -> return $ ifC e1' e2 e3
    (_, Just e2', _) -> return $ ifC e1 e2' e3
    (_, _, Just e3') -> return $ ifC e1 e2 e3'
    _                -> Nothing
```

```
evalEqual :: Fix LNG -> Fix LNG -> Maybe (Fix LNG)
evalEqual (LitP x)  (LitP y)  = return . boolV $ x == y
evalEqual (BoolP x) (BoolP y) = return . boolV $ x == y
evalEqual e1        e2        = case (eval e1, eval e2) of
    (Just e1', _) -> return $ equal e1' e2
    (_, Just e2') -> return $ equal e1 e2'
    _             -> Nothing

evalLam :: Type -> Fix LNG -> Maybe (Fix LNG)
evalLam t e = case eval e of
    Just e' -> return $ lam t e'
    _       -> Nothing

evalApp :: Fix LNG -> Fix LNG -> Maybe (Fix LNG)
evalApp (LamP _ e) e2 = return . shift (-1) 0 $ subst 0 (shift 1 0 e2) e
evalApp e1         e2 = case (eval e1, eval e2) of
    (Just e1', _) -> return $ app e1' e2
    (_, Just e2') -> return $ app e1 e2'
    _             -> Nothing

eval :: Fix LNG -> Maybe (Fix LNG)
eval (LitP _)       = Nothing
eval (AddP x y)     = evalAdd x y
eval (MulP x y)     = evalMul x y
eval (BoolP _)      = Nothing
eval (IfP x y z)    = evalIf x y z
eval (EqualP x y)   = evalEqual x y
eval (VarP _)       = Nothing
eval (LamP t x)     = evalLam t x
eval (AppP x y)     = evalApp x y

subst :: Int -> Fix LNG -> Fix LNG -> Fix LNG
subst n x (VarP v)   = if v == n then x else var v
subst n x (LamP t e) = lam t $ subst (n + 1) (shift 1 0 x) e
subst n x (In e)     = In . fmap (subst n x) $ e

shift :: Int -> Int -> Fix LNG -> Fix LNG
shift i c (VarP n)   = if n < c then var n else var (n + i)
shift i c (LamP t e) = lam t $ shift i (c + 1) e
shift i c (In e)     = In . fmap (shift i c) $ e

evaluate :: Fix LNG -> Fix LNG
evaluate e = case eval e of
    Just e' -> evaluate e'
```

```
    _       -> e
```

C.4 Checking if there are bounded variables

Below code checks if there are bounded variables in `Fix LNG` expressions.

```
class BoundAlg (f :: * -> *) where boundAlg :: Alg f (Int -> Bool)
instance (BoundAlg f, BoundAlg g) => BoundAlg (f ⊕ g) where
  boundAlg (Inl x) = boundAlg x
  boundAlg (Inr x) = boundAlg x

instance BoundAlg LitF   where boundAlg _ _ = True
instance BoundAlg AddF   where boundAlg (Add x y) n = x n && y n
instance BoundAlg MulF   where boundAlg (Mul x y) n = x n && y n
instance BoundAlg BoolF  where boundAlg _ _ = True
instance BoundAlg IfF    where boundAlg (If x y z) n = x n && y n && z n
instance BoundAlg EqualF where boundAlg (Equal x y) n = x n && y n
instance BoundAlg VarF   where boundAlg (Var x) n = x <= n
instance BoundAlg LamF   where boundAlg (Lam _ x) n = x (n + 1)
instance BoundAlg AppF   where boundAlg (App x y) n = x n && y n

isBounded :: Fix LNG -> Bool
isBounded e = fold boundAlg e (-1)
```