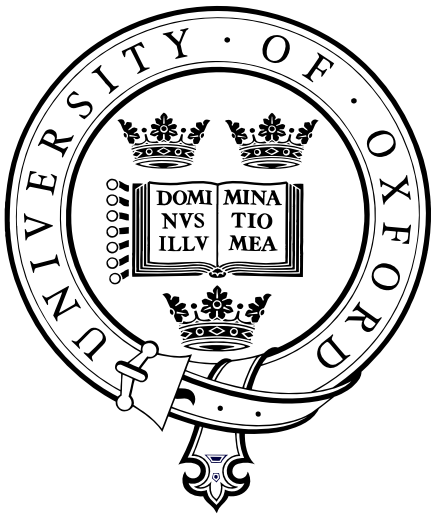
Genericity, extensibility and type-safety in the VISITOR pattern

Bruno César dos Santos Oliveira

Wolfson College



Oxford University Computing Laboratory

Submitted for the degree of Doctor of Philosophy

Abstract

A *software component* is, in a general sense, a piece of software that can be *safely reused* and *flexibly adapted* by some other piece of software. The safety can be ensured by a type system that guarantees the right usage of the component; the flexibility stems from the fact that components are *parametrizable* over different aspects affecting their behaviours. *Component-oriented programming* (COP), a programming style where software would be built out of several independent components, has for a long time eluded the software industry. Several reasons have been raised over time, but one that is consistently pointed out is the inadequacy of existing programming languages for the development of software components.

Generic Programming (GP) usually manifests itself as a kind of parametrization. By abstracting from the differences of what would otherwise be separate but otherwise similar specific programs, one can develop a single unified generic program. Instantiating the parameter in various ways retrieves the various specific programs (and ideally some new ones too). Instances of GP include the *generics* (parametrization by *types*) mechanism as found in recent versions Java and C# and Datatype-Generic Programming (DGP) (parametrization by *shape*). Both mechanisms allow novel ways to parametrize programs that can largely increase the flexibility of programs.

Software components and GP, and in particular DGP, are clearly related: GP and DGP provide novel ways to parametrize software, while software components benefit from parametrization in order to be flexible. However, DGP and COP have mostly been studied in isolation, with the former being a research topic among some functional programming communities and the latter being mostly studied within the object-oriented communities.

In this thesis we will argue for the importance of the parametrization mechanisms provided by GP, and in particular DGP, in COP. We will defend that many design patterns can be captured as software components when using such kinds of parametrization. As evidence for this we will, using DGP techniques, develop a component library for the Visitor pattern that is generic (i.e. can be used on several concrete visitors); extensible (i.e. concrete visitors may be extended); and

type-safe (i.e. its usage is statically type checked). A second aspect of this thesis concerns the adaptation of functional DGP techniques to object-oriented languages. We argue that parametrization by datatypes should be replaced by parametrization by visitors, since visitors can be viewed as encodings of datatypes and, through those encodings, the functional techniques naturally translate into an OO setting.

Acknowledgements

I would like to thank Jeremy Gibbons, my supervisor, for two different reasons. Firstly, for being an excellent supervisor, for giving me a very interesting project to work on, for helping me during my first stages of research (when I felt unconfident about my research and abilities), and for always being willing to help me with technical (and bureaucratic) problems. Secondly, for giving me the opportunity to study in Britain and in the beautiful and inspiring city of Oxford, which has greatly helped me on the development of both my personal and professional skills. I cannot forget José Nuno Oliveira for introducing me to the beauty of functional programming with his always interesting lectures and for motivating me into coming to Oxford. Ralf Hinze provided great inspiration for my work and taught me a lot about generic programming. He and Andres Löh were excellent hosts during my one month visit to Bonn, which proved very fruitful in terms of research. My two examiners, Martin Odersky and Ralf Hinze, provided very useful feedback that helped improving the presentation of this thesis substantially. Mike Spivey took me as his student during Jeremy's sabbatical and provided important feedback during my confirmation of status. Richard Bird and Oege de Moor were very helpful in making me focus on a research direction during my transfer examination. The Friday meetings of the Problem Solving Club provided a relaxed environment to discuss research problems and to get feedback about my own research. The meetings of the Datatype-Generic Programming project gave me the opportunity to present progress on my research and to get feedback regularly, and also provided a constant flow of new ideas. The attic crowd (which included, among others, Daniel Goodman, Christopher Aycock, Rui Zhang, Jolie de Miranda, Edward Smith, and Zoltan Miklos) provided an entertaining working environment. Wolfson College and the many friends I made there did an excellent job in my social integration at Oxford. My girlfriend Warnchudee Chalitaporn, who I met at Wolfson, gave me a lot of support and was very patient during my DPhil. Finally, my family, and especially my mother, have always motivated me and gave me the conditions to pursue higher education.

Table of Contents

List of Figures

Chapter 1

Introduction

The *1968 NATO Software Engineering Conference* (Naur and Randell, 1969), is famous for the wide recognition among the community at the time of the so-called *software crisis*. As Edsger Dijkstra's puts it in its seminal paper "*The Humble Programmer*", presented at the *1972 ACM Turing Award Lecture*, the major cause of the software crisis is:

> *that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.* (Dijkstra, 1972)

The causes of the software crisis were directly linked to the overall complexity of the software process and the relative immaturity of software engineering as a profession. The following manifestations of the crisis were identified by the participants of the conference: projects ran over-budget; projects ran over-time; software was of low quality; software often did not meet requirements; projects were unmanageable and code was difficult to maintain.

Nearly 40 years later, is the software crisis a thing of the past? Clearly not! All the manifestations of problems identified at the time are still very pertinent problems in today's software industry. Of course, such an answer begs for the question: does this mean that in 40 years there was no progress at all? As we rightfully may hope, this has not been the case. Indeed, the development of new methodologies; new programming language paradigms; the advent of stronger type systems; as well as the evolution of tools in general, are substantial improvements when we compare

the state of the art of programming at the time with today's equivalent. Still, this is not enough to cope with today's needs. Dijkstra's justification for this remains valid. The basic problem is that:

> *as the power of available machines grew ... society's ambition to apply these machines grew in proportion.* (Dijkstra, 1972)

With the machine's power growing several orders of magnitude in a small number of years, software problems that were just a programmer's dream before become feasible after a few years. What was considered to be a big program in the 70's is, by today's standards a small one; and while using today's tools may be adequate to solve problems of such a size; these same tools have difficulty scaling up to today's big-sized (or even mid-sized) problems.

The way we have been coping with this problem (as well as many others), is by continuously increasing the level of abstraction: from writing machine code directly to programming in current high-level programming languages many abstractions were discovered, which provided us with new ways to reuse software.

1.1 Software Components and Reuse

At the same conference, Douglas McIlroy addressed the audience with a paper entitled "*Mass Produced Software Components*" (McIlroy, 1969). In this paper he set the vision that software should be componentized, i.e. built from prefabricated components: in the same way that a complex piece of electronics is built-up from a set of simpler, standardized smaller pieces, software should be itself built from smaller software components that can be glued together to build more complex programs. Using the analogy with industrial components he argued that the idea of interchangeable parts corresponded to 'modularity' in software engineering; and that the idea of machine tools has an analogue in assembly programs and compilers. His expectations for what software components should provide can be summarized in the following quote from his paper:

> *The most important characteristic of a software components industry is that it will offer families of routines for any given job. . . In other words, the purchaser of a software component from a family will choose one tailored to his exact needs... He will be confident that each routine in the family is of high quality – reliable and efficient. He will expect the routine to be intelligible, doubtless expressed in a higher level language*

> *appropriate to the purpose of the component ... He will expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, he should be able safely to regard components as black boxes.* (McIlroy, 1969)

The first implementation of an infrastructure for this idea, due to McIlroy himself, Pinson and Tague (McIlroy *et al.*, 1978), is the inclusion of pipes and filters into the Unix operating system. The ideas behind pipes and components set the Unix philosophy: "*Write programs that do one thing and do it well.*"; "*Write programs to work together.*"; "*Write programs to handle text streams, because that is a universal interface.*". Out of these three premises only the last one is intrinsic to Unix; the other two can be generally applied to other software problems.

1.2 Component-Oriented Programming

While McIlroy's vision was warmly received by researchers and spawned a wave of enthusiasm that seemed to indicate that component-oriented programming (COP) would soon become mainstream, the truth is that to date that vision has not been fully accomplished. In fact, apart from Unix pipes and a few other successes, component based software development is still the exception rather than the rule. The reasons for this are not trivial to pinpoint. Some researchers argue that current tools and programming languages are not adequate for COP, but it also probably does not help that McIlroy's informal description of a component does not translate into a widely accepted concrete formal definition (Broy *et al.*, 1998).

Object-oriented programming (OOP) has often been regarded as a promising platform for the development of components (Cox, 1990). However, most existing technologies supporting component development are not, in fact, object-based (not at least in the traditional sense). For example, Microsoft COM (Brockschmidt, 1995) does not support subtyping or inheritance. Similarly, the niche component market created around early versions of Microsoft's Visual Basic involved no object-oriented programming either: "*Real objects, as OOP (object-oriented programming) experts rightly point out, rest on the tripod of inheritance, polymorphism, and encapsulation, while VBXes stand only on the single leg of encapsulation*" (Udell, 1994). Pfister and Szyperski argue that objects are not enough and that while COP and OOP do have many things in common, there are some subtle, but important, differences:

> *A programming language is called component-oriented if it provides polymorphism,*

> *information hiding over several objects, late binding and late linking, and type safety... This is in contrast to the typical interpretation of object-oriented programming, which consists of polymorphism, information hiding over individual objects, late binding only, and inheritance (or delegation).*(Pfister and Szyperski, 1996)

The most noticeable absent 'feature' from the definition of a component-oriented language (when compared to an object-oriented language) is inheritance (or delegation). The reason given by the authors is that code inheritance/delegation mechanisms are not sufficiently controllable, allowing us to effectively break encapsulation (for example with the *fragile base class problem*) (Snyder, 1986; Weck and Szyperski, 1996). As Szyperski (2002) mentions a direct consequence of breaking encapsulation is that object-oriented composition does not really work. This is, of course, limiting, since components should be composable.

Perhaps more importantly component-oriented programming requires information hiding over several objects, instead of individual ones. As argued by Pfister and Szyperski, the OOP focus on individual objects is too narrow and often results in software which cannot be used as components. According to their definition, a component is a collection of cooperating objects; which typically implies that the objects are tightly intertwined. They suggest that what is needed is a static and higher-order module-like structuring construct.

1.3 Design Patterns: A Sign of Weakness?

Design patterns, introduced by the '*Gang of Four*' (GoF) (Gamma *et al.*, 1995), are frequently used to abstract non-trivial designs that can be reused in different contexts. A design pattern is a description or template for how to solve a problem that can be used in many different situations. A typical description of a design pattern will have a name, motivation, examples, consequences, implementation trade-offs, and so on. In some sense a design pattern allows us to copy a carefully studied solution for a similar problem. This usually leads (with a correct interpretation of the pattern) to a good implementation. Most design patterns, however, tend not to be captured in some form of reusable software, which seems a step backwards from a software engineering perspective.

Design patterns, as understood by Gamma *et al.* (1995), cannot be considered as components because they do not allow reuse of the patterns as a library, in fact: "*The only reuse patterns provide is reuse of concepts*" (Meyer and Arnout, 2006). A question that can be asked is whether

design patterns can be captured as a more general abstraction. Some authors believe that they cannot and argue that different implementations of the patterns, although similar conceptually, are just incompatible to each other to be captured by a single abstraction. Other authors, however, believe that design patterns can be captured more abstractly, but the problem is that *current programming languages are just too weak* to capture the those abstractions. Norvig (1996) argues that the flexibility of dynamic languages like Lisp allow us to capture most (16 out of 23) of the design patterns in Gamma *et al.* (1995) as reusable code. However, one problem with dynamic languages is that they do not guarantee type-safety, which is considered a key feature of components.

1.4 Datatype-Generic Programming

Generic Programming (GP) usually manifests itself as a kind of parameterization – in particular, the *generics* mechanism (also known as *parametric polymorphism*) found in Java or C# is a particular kind of GP where we have *parameterization by types*. By abstracting from the differences in what would otherwise be separate but similar specific programs, one can make a single unified generic program. Instantiating the parameter in various ways retrieves the various specific programs and, ideally, some new ones too (Gibbons, 2003).

Datatype-Generic Programming (DGP) is another instantiation of GP where programs can be parameterized by *datatypes* (or *type functors*). By a datatype here we mean a container type, which is given a formal semantics via the categorical notion of a *functor*. The *Algebra of Programming* (AoP) movement (Backhouse *et al.*, 1992; Bird and De Moor, 1997) inspires a particular instance of DGP that has a strong formal foundation based on category theory. The work on AoP explored patterns of recursion on different datatypes and showed that we can have a single (generic) program that captures some pattern: as it was demonstrated, patterns of recursion follow the datatype definition. In other words, a generic program to capture one such pattern needs to be parameterized by the shape of the datatype.

It has been argued by Gibbons (2003) that DGP can also be used to capture the abstractions behind many design patterns formally. This would entail several advantages: patterns would be expressible directly as reusable library components; could be reasoned about; and type-checked. In essence, *the abstractions behind design patterns would become true software components*. More recently, Gibbons (2006) substantiated his argument by capturing four of the GoF patterns – con-

cretely, the VISITOR, the COMPOSITE, the ITERATOR and the BUILDER– as higher-order datatype generic programs. The resulting (datatype-generic) programs are parameterized on three different dimensions: "*by the* shape *of the computation, which is determined by the shape of the underlying data, and represented by a type constructor (an operation on types); by the* element type *(a type); and by the* body *of the computation, which is a higher-order argument (a value, typically a function).*" (Gibbons, 2006).

However, there is a problem: while parametric polymorphism (i.e. parameterization by types) and parameterization by functions is available in many languages (one way or another), parameterization by shape is not. Fortunately, in languages like Haskell (Gibbons, 2006; Hinze, 2004) or Scala (Moors *et al.*, 2006) we can *almost* write fully datatype-generic programs (we still need to manually write some boilerplate code, but given this we can define datatype-generic functions). Research languages like PolyP (Jansson, 2000) and Generic Haskell (Löh, 2004) actually have built-in support for some form of datatype-genericity.

1.5 A case study on the VISITOR Pattern

Despite McIlroy's vision for a component-oriented software industry as a solution for the 'software crisis' not being realised yet, recent developments in programming languages (in particular type systems) show some promise that this could happen in the near future. In this thesis we will make the case that DGP can play an important role in COP by allowing many useful abstractions to be captured as *truly* reusable software components. In particular, we will argue that, with type systems supporting some form of DGP, many design patterns could be captured as software components. Moreover, components developed in languages supporting DGP would be — *as they should be* — extensible, flexibly adaptable and, above all, type checkable.

We will show evidence for this by presenting the foundations of a library for the VISITOR pattern — one of the most well-known (and also one of the most complex) design patterns in the GoF book. This presentation will be made using the Scala programming language (Odersky, 2006a) and the end result will be a compilable generic visitor library that can be parametrized by different aspects (including shape) and with programs written using the library having properties that allow them to be reasoned about (assuming a side-effect free setting). We will also show that using our library we can write generic functions (that is, functions that work for any visitor). Furthermore, the library

will allow us to extend visitors with new cases.

1.5.1 The VISITOR: A Valuable Abstraction

How can a thesis be written around a single design pattern? — one may wonder. The *Church numerals*, introduced by Church (1936), can be defined in the untyped lambda calculus as:

$$zero = \lambda f\ x \quad \rightarrow x$$
$$succ = \lambda n\ f\ x \rightarrow f\ (n\ f\ x)$$

This code shall be explained elsewhere in this thesis, but for now it is sufficient to know that its intent is to provide an encoding for the natural numbers as functions. The technique developed by Church to encode numerals can be applied to other datatypes as well: although this code is probably unfamiliar to most object-oriented programmers, it is (arguably) an instance of the VISITOR pattern. The Church numerals played an important role to prove that any computable function can be expressed and evaluated in the lambda calculus. It is interesting that Church's work (which is a very important foundational work in theoretical computer science) and the VISITOR pattern (a highly practical tool available to the object-oriented programmer) are related. It is not clear if the GoF authors knew of this connection, but many researchers have since realised it to an extent that has become folklore knowledge among some communities. Recently Buchlovsky and Thielecke (2005), in work directed to the type-theory community, formalized the relation between visitors and encodings of datatypes precisely and showed the relation of the traditional imperative version of the pattern with a more functional version that is basically a Church encoding. It was partly the inspiration provided by these encodings that led to our development of a VISITOR library (that we present in Chapter 3).

The question that we asked at the beginning of this section can be finally be answered by emphasizing the dual goals of this thesis. The first goal is to exploit how the abstractions that arise in DGP can be useful for COP. The VISITOR pattern is a good example to use because different aspects of it (like extensibility, genericity, and type-safety) can be easily motivated and developed using DGP-related abstractions. The second goal, which aims at exploring good ways to apply DGP techniques in an object-oriented setting, is closely related to visitors. Our reasoning is that since traditional functional DGP techniques are fundamentally connected to datatypes (just recall that we define DGP as parametrization by datatypes) and the 'natural' way to encode datatypes in

object-oriented languages is using visitors then the manifestation of DGP in OO languages should be *parameterization by visitors*.

1.5.2 The Scala Option

Scala was chosen as the programming language used for the development of our visitor library because:

1. *The* VISITOR *(and more generally, design patterns), which is the focus of this thesis, is traditionally associated with object-oriented languages.* Scala is a statically-typed object-oriented language that was designed to remain close to mainstream object-oriented languages. In particular, Scala compiles into the Java Virtual Machine (JVM) and interacts nicely with Java (there has been some work on supporting the .Net platform as well).

2. *Scala is a functional (DGP) language.* Since almost all the previous work in DGP was developed using functional programming languages it was important to use a programming language that supported similar abstractions. Scala supports first-class functions, parametric polymorphism and abstract types, which can be used to encode '*type constructor polymorphism*' (and DGP). These three forms of abstraction satisfy Gibbons' requirements for a DGP language (see Section 1.4).

3. *Scala supports information hiding over several objects.* Unlike traditional object-oriented languages, which only allow information hiding over individual objects, Scala's abstract types together with inner classes allow us to hide information over a set of objects. As discussed in Section 1.2, this is an essential feature of a component-oriented language. From another perspective, it can be said that Scala nicely unifies modules and objects providing what can be seen as a powerful module system.

4. *Scala supports traits and mixin composition.* Unlike current mainstream OO languages which either support single inheritance (with its well-known limitations) or full-fledged multiple inheritance (with its complexity and safety problems), Scala's support for traits and mixins means that the language has a safe, simple and elegant but yet powerful enough mechanism to combine multiple components.

The option not to use a more mainstream OO language such as Java or C# was mainly due to the lack of abstraction provided by the corresponding type systems. In particular, neither language supports (or allows us to encode) fully type-safe forms of DGP. While this issue could possibly be worked around by losing some type-safety, we feel that this would not be desirable because both DGP and COP regard type-safety as a key feature. Moreover, those languages are based on single inheritance and do not support information hiding over several objects, which severely limits their use for COP.

Another option that could be considered would be to use a non object-oriented language that satisfies the DGP criteria. Haskell (Jones, 2003) would be an obvious candidate here since it has been the preferred platform for much of the DGP research in the past. However we feel that the connection to the object-oriented paradigm is important and should be preserved. While it is certainly possible to encode visitors in Haskell (arguably visitors are just typed encodings of datatypes), the fact is that the VISITOR pattern (and many other design patterns) is somehow alien to functional programming languages like Haskell: visitors play the same role as datatypes and it would just be awkward to use visitors when a datatype mechanism is readily available. On the other hand, because typical OO languages do not have datatypes, visitors can be useful to capture abstractions where datatypes would make more sense than an OO design. Therefore, the VISITOR pattern is a valuable abstraction for the OO programmer and not just an intellectual curiosity as it is for the functional programmer. Also, Haskell's weak module support means that information hiding in the large is not well supported, which limits Haskell's use for COP.

A word of honour should go to OCaml (Leroy *et al.*, 2005) that, like Scala, is a functional statically-typed OO language. The option to use Scala instead of OCaml was due to two facts. Firstly, Scala unifies the module system with the object system while OCaml maintains two separate mechanisms. In the authors opinion Scala's solution seems more elegant and easier to use from a programmers perspective. Secondly, Scala remains closer to the traditional OO languages, which makes it easier to argue about and develop design patterns: like Haskell, because OCaml readily supports datatypes, it is somehow awkward to talk about visitors in OCaml; however, unlike Haskell, OCaml has a powerful module system that allows information hiding on modules. Despite these two facts, it should be possible to do a similar development to the one in this thesis using OCaml instead of Scala.

Finally, it should be mentioned that, although we defend that Scala is the most adequate existing programming language for COP and DGP in an OO setting, we believe that there is still a lot of space for improvement. In fact, a secondary goal of this thesis is precisely to explore the strengths and weaknesses of Scala for the development of DGP programs. In Chapter 6 we will discuss our findings and make some suggestions that we hope will be of interest for both the programming languages and COP communities.

1.6 Overview of the Thesis

In Chapter 2 we will introduce the concepts and notation that are going to be used in this thesis. In particular, we will introduce the Scala programming language, present the Visitor and related Composite design patterns and introduce the functional notation used in this thesis to specify parts of our visitor library.

In Chapter 3 we will show how the Visitor design pattern is associated with different encodings of algebraic datatypes (AlgDTs). In particular we explore the connection with Church and Parigot encodings and we will show that, using some type-theoretic results, it is possible to define generic visitor libraries for both encodings. These libraries are essentially parameterized by the *shape* of the concrete visitors. Furthermore, we will add an extra level of parameterization and show that the different kinds of encodings can be themselves a parameter of an even more general library, which is not only parameterized by shape but also by (what we shall call) a *decomposition strategy*. Finally, we will talk about how we could improve the notation for programmers and show how some of that notation can already be implemented within Scala.

The relationship between visitors and AlgDTs leads naturally to a connection with the DGP styles found in functional programming. In Chapter 4 we develop a DGP library for visitors inspired by the '*Generics for the Masses*' (GM) approach proposed by Hinze (2004). With this approach we can define our own generic functions on visitors. We also show that GM is itself one instance of the visitor pattern, but it cannot be implemented with the visitor library presented in Chapter 3. To solve the problem we propose a generalization of our visitor library, which allows us to encode a larger family of visitors (including the one that arises from GM). With the insight that GM can be implemented with visitors, we eliminate the need for a design choice that is present in GM. Finally, we will see how to express a family of visitors based on sums of products within our visitor library,

which allows us to express a wider range of generic functions.

The fact that generic functions cannot be extended is a severe drawback, because often we want to define some ad-hoc behaviour for new datatypes. This limitation precludes the design of an extensible and modular generic programming library. In Chapter 5 we will talk about the the *expression problem* (Wadler, 1998) and how it relates to the extensibility problem of generic functions. We will then show how we can make our visitors (and generic functions) extensible. Two different solutions will be presented. The first solution will allow only extensible Church encodings, but it will be easier to use than the second solution, which in turn will allow any encoding. Finally, we will discuss the different trade-offs of the two solutions.

In Chapter 6 we will start by summarizing our results and contributions and discussing some extra insights that we found worth mentioning. We then briefly present the results of this thesis from a type-theoretic perspective. After that, we compare Haskell and Scala for the implementation of visitor and DGP libraries, discuss their trade-offs and show some important shortcomings of Haskell. Finally, we discuss some applications of our work and propose some future work.

1.7 Related Work

1.7.1 Encodings of Datatypes and the Visitor Pattern

An important foundation of this thesis regards the connection between encodings of datatypes (such as Church encodings) and the Visitor pattern. As mentioned earlier, many researchers have realised this connection and the work by Buchlovsky and Thielecke (2005) establishes this relation more formally by reconstructing the Visitor pattern in Java (with support for generics) from a Church encoding in a minor variant of System F_ω.

The Church numerals (along with encodings of other datatypes) were first presented by Church (1936) in the untyped lambda calculus. Böhm and Berarducci (1985) demonstrated that in System F it is possible to give precise typings to those encodings. The name '*Church encoding*' is normally associated with Böhm and Beraducci's System F encoding. There are well-known limitations on the expressiveness of Church encodings (certain functions that are inherently inefficient or even inexpressible). A less well-known encoding is the *Parigot encoding* (Parigot, 1992), which basically allows us to write any *generally recursive* definitions (in contrast to the Church encoding). However this encoding requires System F to be extended with recursion. Jan Martin Jansen (2005)

uses, perhaps unawaringly, what is essentially an untyped version of the Parigot encoding to show how to translate datatypes and pattern-based function definitions systematically.

Buchlovsky and Thielecke exploit the relationship between Visitors and encodings of datatypes. They define two different categorizations of visitors with two possible choices in each. The first categorization distinguishes who controls the traversal: a visitor is called *internal* when the visitable classes define the traversal behaviour, and *external* when that role is performed by the visitor itself. The second categorization is between *functional* and *imperative* visitors. The difference is that the *visit* methods of the former kind of visitor can be seen as pure functions (a result is returned) whereas in the later they make use of internal state to store results (thus, no result is returned). Their paper is focused mostly on internal visitors (since external visitors in their idealized System F_ω are not very useful due to the lack of built-in recursion) and discusses how we can derive both functional and imperative versions of visitors.

The most common rendering of the Visitor pattern is imperative (with both the internal and external alternatives being common). In this thesis we discuss a functional model instead, with both internal and external variations being covered. Buchlovsky and Thielecke's work becomes relevant for us because they formally show the equivalence between functional and imperative visitors — albeit in a somewhat idealized setting. We use their work to partly justify our option of using functional instead of imperative visitors: since some formal equivalence exists we are free to use either option.

1.7.2 Design Patterns and Components

By capturing the knowledge and experience of software designers, design patterns prove to be a valuable resource for software design. However, from a software engineering perspective, capturing design patterns as prose instead of some reusable component seems a step backwards. Some researchers think that the reason for this is that design patterns are simply not componentizable. Other researchers, including us, disagree and point the finger to the lack of abstractions in current programming languages.

Odersky and Zenger (2005b) point out that the Scala programming language is designed with component development in mind. In their work they identify *abstract type members*, *self type annotations* and *modular mixin composition* as abstractions that do not exist in mainstream OO languages but prove to be important for component development. Using the first two features they

provide an elegant software component that captures the OBSERVER design pattern; all these features are later used in the larger scale case study about the design of the Scala compiler.

Scala is probably the closest to a mainstream language that is developed with COP in mind, but there are other research languages that share similar goals or have abstractions that could support COP. The CaesarJ programming language (Aracic *et al.*, 2006) provides a single construct that unifies aspects, classes and packages. This construct is powerful enough to solve different problems from aspect-oriented and component-oriented programming. The GBeta programming language (Ernst, 1999) has powerful constructs like virtual classes, a general block structure and dynamic multiple inheritance, which proves to be useful for COP.

Arnout (2004) reviewed all the 23 patterns described in Gamma *et al.* (1995) and evaluated their componentizability in the Eiffel programming language (Meyer, 1997). The results showed that about two thirds of the design patterns could be replaced by a corresponding component; a quarter of the patterns had "Wizard or library support"; and the remainder (2 out of 23 patterns) were classified as non-componentizable. According to Arnout, Eiffel features like *genericity*, *tuples* and *agents* played an essential role in the componentization of design patterns.

The VISITOR as a Component There have been several proposals for *generic visitors* (visitor libraries that can be reused for developing software using the VISITOR pattern) in the past. Palsberg and Jay (1998) presented a solution relying on the Java reflection mechanism, where a single Java class *Walkabout* could support all visitors as subclasses. Refinements to the idea of using reflection to capture generic visitors, mostly to improve performance, have been proposed since by Grothoff (2003) and Forax *et al.* (2005). One advantage of these approaches is that they are not invasive — that is, the visitable class hierarchies do not need to have *accept* methods, which adds some flexibility and extensibility because there is no need to plan software with visitors in mind initially. However, the heavy reliance of the approaches on reflection hinders type-safety; because of that, those solutions should not be strictly classified as components (at least according to our definition of a component).

Arnout, jointly with Meyer (Meyer and Arnout, 2006), showed how to componentize the VISITOR pattern. Like the previous proposals for generic visitors, they define a non-invasive version of the VISITOR; however their visitor is less reliant in introspection mechanisms (although these are still needed), which makes the approach more type-safe. The proposed component consists of a single generic *Visitor* class that can be reused by concrete visitors by parametrizing the class with

the concrete visitable interface. In essence, *Visitor* is implemented as a collection of *visit* agents (which are roughly analogous to delegates or closures) and provides a *visit* method that can be called by the clients. Since the *Visitor* is parameterized by the visitable interface, only agents that have as an argument a subtype of that interface can be added into the collection; if we try to add an agent that does not satisfy this we will get a type error. When the *visit* method is called, an introspection mechanism is used to determine at run-time which agent should be called on the *visit* argument. Performance measurements indicate that using this component is only slightly slower than traditional visitors, but has the big advantage of reuse.

1.7.3 Design Patterns and Functional Programming

Norvig (1996) was among the first to explore the relationship between design patterns and functional programming. In his study he used the Lisp and Dylan programming languages, both of which are (functional) dynamically typed languages and support some kind of object system. In his study he classified design patterns as *invisible* (the programming language supports abstractions that eliminate the need for the design pattern), *informal* (the design pattern can only be captured as prose) and *formal* (it is possible to capture the design pattern formally in the language). What his study revealed was the flexibility of dynamic languages such as Lisp or Dylan allowed 16 out of 23 of the design patterns in Gamma *et al.* (1995) to have qualitatively simpler implementations — either because they were invisible or they could be formalised.

Kühne (1999) argues that *"Design patterns inspired by functional programming can advance object-oriented design"* and develops several design patterns from that inspiration. From the opposite perspective, Läufer (2003) asks *"What functional programmers can learn from the visitor pattern"* and argues that functions in functional programming languages are too inflexible since they do not allow reuse by inheritance. Using the connection between visitors and datatypes and inspired by existing encodings of inheritance in functional languages, he proposes a simple technique, which can be used in functional languages, that allows programmers to encode reuse by inheritance of functions. We believe that inspiration can be drawn from both sides and we hope to be able to contribute some new insights to both functional and object-oriented programmers.

Gibbons (2003) argues that DGP can be used to capture the abstractions behind many design patterns formally. There would be several advantages in doing so: patterns would be expressible directly as reusable library components; could be reasoned about; and type-checked. In essence, the

abstractions behind design patterns would become true software components. He substantiates his position in Gibbons (2006) by arguing that four of the design patterns in GoF are effectively related to recursion patterns that had been developed by the Algebra of Programming movement. Specifically, he argues that the VISITOR corresponds to *folds* (or *catamorphisms*); the ITERATOR corresponds to *maps*; the BUILDER corresponds to *unfolds* (or *anamorphisms*); and COMPOSITES corresponds to datatypes. However, as he notes, these comparisons are made under simplifying assumptions such as taking a specific interpretation of the design pattern. For example, the ITERATOR is related to maps only when interpreted as INTERNAL ITERATOR and there is the assumption of no side effects. The later simplification has been lifted by Gibbons and Oliveira (2006) where it is argued that *applicative functors* can be used to model side-effects that exist in imperative languages; and the corresponding *traverse* operation (an *effectful* form of map) models internal iteration with effects.

The Algebra of Programming In his work on the relation between DGP and design patterns, Gibbons was inspired by the Algebra of Programming movement (Malcolm, 1990; Meijer *et al.*, 1991; Backhouse and Hoogendijk, 1993; Bird and De Moor, 1997). This movement works on a branch of the mathematics of program construction that studies the relationship between the structure of programs and the structure of the data that they manipulate.

The practical results of this line of research are the so-called 'recursion patterns': functions that capture common structures of programs using datatypes. The interesting thing about these patterns is that they do not depend on specific datatypes; instead they can be parameterized on the datatypes themselves. For example, the most well-known of these is the *fold* function, which basically captures definitions by structural recursion on the shape of datatype. Although the most common instantiation of *folds* is with lists, similar operations still make sense with any tree-like structures. Therefore, instead of providing separate *fold* functions for each datatype, we can provide a single *datatype-generic* definition.

On the theoretical side, this line of research is based on the formal foundations of category theory, which provides a solid ground for reasoning about programs written using recursion patterns. Using the categorical setting we can derive properties and programs using an equational style that resembles secondary school algebra. One practical benefit (and a major motivation) is that efficient programs can be derived from less efficient ones by just using equational reasoning. In particular, *fusion* laws play a crucial role in this optimization process by telling us how can we fuse programs in such a way that they take a single traversal over the data.

1.7.4 Functional DGP

There are two main streams of work on functional DGP (that is, DGP in the context of functional programming languages). The first (and earlier) stream of work, which we shall refer to as *traditional* DGP, involves the development of new languages or non-trivial language extensions that allow native language support for DGP. The second (and more recent) stream of work, which we shall refer to as *lightweight* DGP, usually builds on existing language features to provide some form of library support for generic programming. There are some in-between forms of DGP that require only a relatively mild modification on the programming language (Hinze and Peyton Jones (2000) is one example). In this thesis we are particularly interested in lightweight DGP; in Chapter 4 we develop the basis of a lightweight DGP library in Scala.

Traditional DGP

PolyP *PolyP* (Jansson, 2000) is a language extension to Haskell, that allows the definition of generic functions over regular datatypes of kind $* \rightarrow *$. Generic programming in PolyP is based on the notion of *pattern functors*. Each datatype is associated with a pattern functor that describes the structure of the datatype. Isomorphisms between pattern functors and the actual datatype are automatically generated by PolyP and can readily be used in programs. In order to define generic functions a special construct *polytypic* is used, which allows the function to exploit the shape of the pattern functor in its definition. In PolyP, Algebra of Programming style recursion patterns like *folds* can be defined truly generically (i.e. requiring a single definition only and with no additional boilerplate code). The original PolyP translated polytypic definitions to Haskell using a specialization approach. In the more recent PolyP2 (Norell and Jansson, 2004), type classes and functional dependencies are used in order to perform the translation.

Generic Haskell *Generic Haskell* (GH) (Löh, 2004; Löh *et al.*, 2005) is a generic programming extension to Haskell. GH addresses some of the limitations of PolyP; in particular it allows the definition of generic functions over a much wider range of datatypes (nearly all Haskell 98 datatypes are within the range). This contrasts with PolyP, which imposes severe limitations: only kind $* \rightarrow *$, only regular (not nested) datatypes and also no mutually recursive datatypes. The key idea is to exploit the fact that Haskell 98 datatypes are algebraic and, therefore, they can be perceived as nested binary sums of products. The sums of products are to GH what pattern functors are to

PolyP and, like PolyP, an isomorphism between datatypes and sums of products is automatically generated. Generic functions are defined structurally on sums of products. Unlike PolyP, it is not possible to define recursion patterns in GH, which is an important limitation. This limitation arises from the fact that sums of products loose the information about recursion points. Some work has been done by Holdermans *et al.* (2006) to alleviate this and some other related problems.

Type-indexed datatypes GH also supports *type-indexed datatypes*, allowing us to encode advanced forms of generic programming; for example, a generic version of the Zipper (Huet, 1997) is implemented in Hinze and Jeuring (2001). Recent work by Chakravarty et al. (Chakravarty *et al.*, 2005b,a) proposes related extensions to Haskell. The idea is to extend type classes to support not only ad-hoc overloading — or type-indexing — on functions but also on *types* and *datatypes*. Some applications include those of type-indexed datatypes, self-optimizing libraries that adapt their data representations, and algorithms that work in a type directed manner.

Lightweight DGP

Type Classes Haskell's type class system is an (ad-hoc polymorphism) mechanism that allows the definition of type-overloaded definitions. A primitive generic programming mechanism exists already in Haskell 98 using type classes. The so-called '**deriving**' mechanism automatically derives implementations of commonly overloaded functions such as *equality* or *pretty printing*. However this mechanism only works for a few, built-in, type classes, and extending it is not possible. Hinze and Peyton Jones (2000) propose a simple extension to the type class mechanism that would address the limitation of the **deriving** mechanism and thus allow support for generic programming directly in Haskell. The key idea is to write default method definitions in a class declaration that exploit a sum-of-products-like structure of datatypes. Like GH and PolyP, all the isomorphisms between the sums of products and datatypes are automatically handled by the compiler. One limitation of this mechanism is that it only works on types of kind $*$. Clean's generics system (Alimarine and Plasmeijer, 2001) generalizes derivable type classes to allow generic type classes that can be defined at arbitrary kinds rather than just $*$. This is achieved by generating a (possibly) infinite set of classes, one class per kind.

Scrap your boilerplate Lämmel and Peyton Jones (2003) presented 'Scrap your Boilerplate' (SyB): an approach to generic programming based on a simple type-safe cast operator, which makes

it is possible to (dynamically) compare two types in order to determine if they are (nominally) equal. The generality available is thus created by extending polymorphic, uniform traversal functions with type-specific behaviour using this cast operator. A library of traversal combinators that works on any *Typeable* datatype was developed and is available in recent versions of the GHC Haskell compiler. The traversal combinators include top-down and bottom-up traversals and queries. In Lämmel and Peyton Jones (2004) introspection facilities were added to the library, which allowed the development of generic functions that made use of meta-data such as constructor names or their arity. In this work it was also shown how to encode generic zips (which seemed tricky to achieve in their earlier work). In Lämmel and Peyton Jones (2005) the issue of extensibility of generic functions was addressed by providing a completely new implementation of SyB that used type classes instead of the type-safe cast operator to model type-specific behaviour — since the type-safe cast operator implied that all type-specific cases needed to be provided at once.

Hinze *et al.* (2006) presented one alternative implementation for a SyB-like library. The main contribution of this implementation was the so-called *spine-view*, which provided a way to access the structure of datatypes (lacking in the previous SyB implementations). This contribution was important in comparing the SyB approach to generic programming with other approaches. With the insights gained it was shown that SyB could be applied to a very large class of data types that included, for example, some GADTs (see below). In subsequent work, Hinze and Löh (2006) introduced the so-called *type-spine view* to allow the definition of consumer functions; and in Hinze and Löh (2008) a detailed study, locating the SyB approach in the design space of generic programming, was presented.

Bringert and Ranta (2006) present a line of work that seems to be closely related to the SyB approach and shows an alternative technique that can be used to remove boilerplate code. Using this technique, the programmer has to define once a generic part per each datatype (the *compos* function) and then he can use that function to define boilerplate operations over the values of the datatype. The *compos* function, in its general form, takes two arguments that have, essentially, the same types as the applicative functor McBride and Paterson (2007) operations. Due to this connection we could use the laws of applicative functors to prove properties about functions written with *compos*. Another interesting aspect of this work is that the technique can be used in languages other than Haskell. For example, the authors show how by using parametrized visitors the pattern could be used in Java.

Encoding Type Representations *Generalized algebraic data types* (GADTs) (Peyton Jones *et al.*, 2006), also known as "*guarded recursive data types*" (Xi *et al.*, 2003) or "*first-class phantom types*" (Cheney and Hinze, 2003), are a generalisation of algebraic datatypes supported by some recent versions of functional languages. GADTs allow the programmer to define a form of *type-indexed datatypes* and can be used to enforce some typing constraints. Hinze (2003) shows how to use *phantom types* to define type representations which can then be used to define generic functions over the represented types. In earlier work (Cheney and Hinze, 2002), type representations are encoded making use of existential types and a type equality operator instead of having to rely on the availability of GADTs.

Hinze (2004) GM approach shows how to encode generic functions within Haskell 98. Once again, the idea is to provide a type representation, but type representations are encoded with type classes instead of datatypes. A generic function can be encoded as an instance of class *Generic*. Another class (the *Rep* class) defines a function *rep* which can be used to construct the type representations automatically. The generic library proposed by Hinze comes in two different flavours, which provide two slightly different interpretations of generic functions. The inspiration for these two different flavours comes from encodings of datatypes (specifically, the Church and Parigot encodings), which are greatly explored in this thesis (see Chapter 3).

One limitation of most approaches based on type representations is that generic functions are not extensible, which severely hinders modularity. Oliveira *et al.* (2006) shows a variation of GM that solves this problem by generalizing the *Rep* class. In Chapter 5 we will make use of this work to show how we can achieve one extensible version of our Visitor library. Weirich (2006) proposes 'RepLib', which is a generic programming library using type representations encoded with GADTs that is also extensible. The extensibility is achieved using type classes.

1.7.5 The Expression Problem

The term '*expression problem*' was originally coined by Wadler (1998) (although the problem was previously known). According to Wadler, a solution for the problem should allow the definition of a datatype with the addition of both new variants and functions being possible. Furthermore, a solution should not require recompilation of existing code, and it should be statically type safe: applying a function to a variant for which that function is not defined for should result in a compile-time error. Odersky and Zenger (2005a) add '*independent extensibility*' (it should be possible to

combine independent extensions) to the list of criteria of what constitutes a solution to the expression problem. The expression problem plays an important role in COP since *"extensible systems are in principle modular, have no final form or final integration phase, cannot be subjected to final total analysis, cannot be exhaustively tested, and have to allow for mutual independence of extension providers"* (Szyperski, 1996). In Chapter 5 we show that our visitor components can be made extensible in both dimensions (functions and variants), therefore not suffering from the expression problem.

Generics Wadler's solution for the expression problem in Generic Java, showed how generics could be used to solve the problem using an encoding of self types. Unfortunately, his solution was later found to be unsound. Four different solutions using generics in Java and/or C# were presented by Torgersen (2004). The first two solutions work in both Java and C#, while the third solution relies on Java wildcards and the fourth solution relies on dynamic reification of type parameters that is only present in C#. Because of the dependency of the fourth solution on dynamic reification, it does not totally satisfy the type-safety requirements of a solution to the expression problem. Torgersen also defined some terminology that is useful to compare different solutions.

Haskell There is a folklore solution to the expression problem in Haskell. The key idea consists in lifing all the variants of the "open datatypes" to datatypes and then use type classes to define the functions on that datatype. In this solution, the type class defines a method f which represents the open function we want to define and each type class instance corresponds to a case (on a variant) of f. However this approach has important limitations in the kinds of functions we can express: n-ary functions (or methods); nested case analysis and mutually recursive definitions are all problematic to express. Swierstra (2008) recently proposed an elegant approach combining folds and type-level extensible sums. With this approach, open datatypes can be compositionally written and functions over those datatypes can be elegantly defined using type classes. However, dispite the gain of clarity and elegance, the same limitations as the folklore solution apply.

Polymorphic variants Garrigue (2000) shows how *polymorphic variants* in OCaml can be used to solve the expression problem. With polymorphic variants, different datatypes can share the same constructor. When a definition using pattern matching is written every usage of a polymorphic variant will raise a type constraint, which ensures that only a datatypes containing all of those

constraints will be used in the definition. A limitation of this approach is that only top-level pattern matching is supported. In subsequent work, Garrigue (2004) addresses this problem and shows a solution for typing pattern-matching in the presence of polymorphic variants. However his solution has to make some compromises to achieve this. In particular the strategy used does not guarantee that all polymorphic variant pattern-matching is complete, since enforcing this would require a very restricted type. Therefore, Garrigue's solution for the deep pattern-matching problem is not type-safe in Wadler's sense.

Extensible algebraic datatypes with defaults Zenger and Odersky (2001) propose *extensible algebraic datatypes with defaults* as a possible solution for the expression problem. They observe that the subtyping relationship between a datatype and its extension is inverted (the extension is a supertype of the original datatype), which leads to the idea of adding a default variant to every algebraic datatype. This has the effect of subsuming all variants defined in future extensions of the type. Unlike traditional datatypes, in this alternative model the extension is a subtype of the original datatype. This solution is, however, subject to single inheritance, which means that only linear extensions are possible. Moreover, it assumes that sensible default cases exist for all functions, which may not necessarily be the case.

Virtual Types Odersky and Zenger (2005a) present two solutions for the expression problem using a combination of abstract types and nested classes. In the top-level classes, some operations and variants are initially added and the hard references that would preclude extensibility are replaced with abstract types. In the subclasses, new operations and/or variants can be added by suitably extending the top-level class and refining the abstract types. Their solution has, somehow, the flavour of *virtual classes*, which provide a more direct way to solve the problem. A solution for the expression problem using virtual classes is presented, for example, by Ernst (2004) in the GBeta programming language. Ernst's solution also benefits from a special composition operation that can compose two classes and all of its inner classes automatically. In Scala we have to perform this operation manually. Nystrom *et al.* (2004) present a solution in Jx that is very similar to Ernst's one, however instead of virtual classes they use a slightly different mechanism that does not suffer from the unsoundness problems usually associated with virtual classes. Jx also supports *nested inheritance*, which is similar to the composition operation in GBeta allowing both the classes and their inner (or nested) classes to be automatically composed.

Chapter 2

Preliminaries

In this chapter we will introduce some material in order to keep this thesis self-contained. However we will assume familiarity with both functional and object-oriented programming. In Section 2.1 we will introduce the Scala programming language. In Section 2.2 we will show how Scala can be used as a DGP language. In Section 2.3 we will present the Visitor pattern and the related Composite pattern. Finally, in Section 2.4 we will informally present the functional notation that we will use throughout this thesis to specify some of our components and reason about them.

2.1 The Scala Programming Language

Scala is a strongly typed programming language, built on top of the JVM, that combines object-oriented and functional programming features. Although strongly inspired by recent research, Scala is not just a research language; it is also aimed at industrial usage: a key design goal of Scala is that it should be very easy to interact with Java, making huge amounts of Java libraries readily available for programmers. The user base of Scala is already quite significant, with the compiler being actively developed and maintained. In this section we will provide an introduction to Scala concepts. For a more complete introduction/description of Scala, see Odersky (2006a, 2007a,b); Schinz (2007).

2.1.1 Expressions and Definitions

Definitions In Scala, we can define functions using the **def** keyword. For example, a function that squares a double could be defined as:

def *square* $(x : double) : double = x * x$

This declaration reads as follows: define a new function *square* that takes an argument x of type *double* and returns a *double* computed by $x * x$.

Values When we execute a definition **def** $x = e$, the expression e will not be evaluated until x is used. If we wish to evaluate the right-hand-side e as part of the evaluation of the definition, Scala offers a value definition **val** $x = e$ which provides this behaviour. One important difference between values and definitions is that only definitions can take parameters (values are just constants).

Conditional Expressions Scala's syntax for conditionals is similar to Java but, unlike Java, it can be used not only between statements but also between expressions. This means that it serves as a substitute for Java's conditional expressions ... ? ... : ..... Here is one example:

def *abs* $(x : double) : double =$ **if** $(x \geqslant 0)\ x$ **else** $-\ x$

First-Class Functions In Scala functions are '*first-class values*'. Therefore, we can define *higher-order functions*. For example, here is how to define the function *twice* that, given a function f, applies f twice to its argument x.

def *twice* $(f : int \Rightarrow int, x : int) : int = f\ (f\ (x))$

Scala supports *anonymous functions*. For instance, to define a function that raises an integer to the power four, we could use the function *twice* together with one anonymous function to achieve that effect. Here is how:

def *power4* $(x : int) : int =$ *twice* $((y : int) \Rightarrow y * y, x)$

The first argument of the function twice is an anonymous function that takes an integer y and returns another integer $y * y$.

Scala also supports *currying*. To declare a curried function we can use two different pieces of syntax. Here are two examples:

def *twiceCurry* $(f : int \Rightarrow int)\ (x : int) : int = f\ (f\ (x))$

```
def comp : (int ⇒ int) ⇒ (int ⇒ int) ⇒ int ⇒ int =
  f ⇒ g ⇒ x ⇒ f (g (x))
```

The first example is just the curried version of the function *twice*, where the first and second arguments are named *f* and *x*. In the second example we present a composition operator on functions that take integers and return integers. In this example the arguments are not named. Either syntactic option can be used to define curried functions, and we can even mix the two notations; for example

```
def twiceCurry2 (f : int ⇒ int) : int ⇒ int = comp (f) (f)
```

would name the first argument *f* but have an anonymous second argument — alternatively one may think of *twiceCurry2* as a function that takes a function and returns a function.

Infix Operators In Scala an *infix operator* can be any one argument operator. As a simple example, consider the following class:

```
class NatInt (x : int) {
  val value = x
  def is (y : NatInt) : boolean = value.equals (y.value)
  def isZero : boolean = this is zero
  def + (x : NatInt) : NatInt =
    new NatInt (this.value + x.value)
}
```

In this class we have a couple of definitions that can be used as infix operators. The + definition, for example, can be used to add two naturals by either writing *n1 .+ (n2)* or *n1 + n2*. We may be tempted to think that this stems from the fact that + is a symbolic and not an alphabetic identifier. However this is not the reason why + can be used as an operator. In fact, the definition *is* can itself be used as an operator because it takes one argument. So we can either write the definition of *isZero* as **this**.*is* (*zero*) or as **this** *is zero*. The only difference between using an alphabetic and a symbolic identifier is that Scala gives them different priorities when used as operators.

Lazy Arguments Arguments can be passed by name by prefixing the type with ⇒. Lazy arguments are specially useful when defining our own control structures. Scala's parsing combinators are a good example of the use of laziness. For example, the combinator &&& which applies a parser and if that parser succeeds applies another parser is declared as:

```
def &&& (q: ⇒ Parser) : Parser = ...
```

In this example, the argument *q* (the second parser) is lazy. This means that only if *q* is needed it will be evaluated.

2.1.2 Classes and Objects

Classes In Scala, classes with parameterless constructors are declared similarly to Java.

```
class Talker {
  def talk (str : String) : Unit = System.out.println (str);
}
```

In this example the *Unit* type plays the same role as *void* in other languages.

When it comes to constructors with parameters, Scala differs from other mainstream languages. Scala encourages a single constructor per class by using the following syntax:

```
class TalkerPar (str : String) {
  def talk () : Unit = System.out.println (str)
}
```

A class can *inherit* from another class using the **extends** keyword. As an example, consider a new class *TwiceTalker* that inherited from *Talker* and added two new methods to it. We could do this by:

```
class TwiceTalker extends Talker {
  private def doTwice (f : String ⇒ Unit, x : String) : Unit = {f (x); f (x)}
  def talkTwice (str : String) : Unit = doTwice (talk, str)
}
```

Note that, like in Java or C#, we can have private members by using the **private** keyword. In this example the *doTwice* method is private.

Overriding methods in Scala requires the use of an explicit **override** keyword. The next example overrides the method *talkTwice* from the *TwiceTalker* class:

```
class TwiceShouter extends TwiceTalker {
  override def talkTwice (str : String) : Unit =
    super.talkTwice (str.toUpperCase ());
}
```

The syntax for the creation of objects is fairly standard. Here are a couple of examples instantiating and interacting with some of the classes that we have been defining:

```
scala > def u = new Talker ();
scala > u.talk ("Ola");
Ola
scala > def p = new TalkerPar ("Hola");
scala > p.talk ();
Hola
scala > def t = new TwiceTalker ();
scala > t.talkTwice ("Hello");
Hello
Hello
```

Abstract Classes Like in other OO languages, we can have *abstract classes* (that is, classes where some methods may be undefined). To declare an abstract class we append the **abstract** keyword to the class declaration:

```
abstract class AbstractTalker {
  def talk (str : String) : Unit
}
```

In this case, the method *talk* is abstract. Unlike in other languages, no **abstract** keyword is needed in method declarations.

Objects When a class has a single instance, we can avoid the creation of a new object each time we want to use that class by using an **object**. Object definitions follow the syntax of class definitions; they can have optional **extends** clauses and a body. However, unlike classes, with object definitions we cannot create other objects using **new**. Furthermore, object definitions do not have constructor or type parameters. Here is an example of an object followed by a definition which calls a method of that object.

```
object HowDoYouDo extends TalkerPar ("Hello!") {
  override def talk () = {super.talk (); System.out.println (" How do you do?"); }
}
def greet = HowDoYouDo.talk ()
```

Case Classes Scala supports the notion of *case classes*, which provide some syntactic sugar and allow the definition of functions by case analysis. With case classes we can emulate AlgDTs from conventional functional languages. For example, we could define a simple hierarchy of classes for defining the Peano numerals as follows:

```
abstract class Nat
case class Zero extends Nat
```

case class *Succ* (*n* : *Nat*) **extends** *Nat*

The first class declares an abstract class *Nat*, which is the supertype of the natural numbers. The class *Zero* acts as the base case and the class *Succ* can be built provided a natural number *n*. Note that if we do not need to define any methods in a class, we could skip writing the empty body {}.

We can also define functions by case analysis on the naturals. For example,

```
def nat2int (n : Nat) : int = n match {
  case Zero ()  ⇒ 0
  case Succ (m) ⇒ 1 + nat2int (m)
}
```

defines a function that converts the natural number into a built-in integer.

Case classes also benefit from an automatically defined constructor function (with the same name as the class), which allows us to write

def *three* = *Succ* (*Succ* (*Succ* (*Zero*)))

instead of the more longwinded version:

def *threeLong* = **new** *Succ* (**new** *Succ* (**new** *Succ* (**new** *Zero* ())))

2.1.3 Traits and Mixins

Traits Scala has a special kind of abstract classes called *traits* (Schärli *et al.*, 2003). Like abstract classes, traits can have abstract methods and defined methods. The difference is that traits cannot have parameters but they can be *mixed in* together. With *mixin composition* a kind of safe *multiple inheritance* is possible. The next example demonstrates the use of traits in Scala:

```
trait Hello {
  val hello = "Hello!"
}
trait HowAreU {
  val howAreU = "How are you?"
}
trait WhatIsUrName {
  val whatIsUrName = "What is your name?"
}
trait Shout {
  def shout (str : String) : String
}
```

Mixin composition As we can see traits can be used much like abstract classes, allowing both the declaration of both abstract and concrete methods. Mixin composition solves the problem of multiple inheritance, while allowing similar expressive power, by ensuring that it is possible to linearize the inheritance relationship. Next we show how we could combine the traits of our example using Scala's mixin composition:

```
trait Basics extends Hello with HowAreU with WhatIsUrName with Shout {
  val greet = hello + "  " + howAreU
  def shout (str : String) = str.toUpperCase ()
}
```

The trait *Basics* inherits the methods from *Hello*, *HowAreU* and *WhatIsUrName*; implements the method *shout* from *Shout*; and defines a new method *greet* that uses the inherited methods *hello* and *howAreU*.

2.1.4 Generic Types and Methods

Parametric Polymorphism Like Haskell or ML (and more recently Java and .Net), Scala supports *parametric polymorphism*. For example, the function *comp* that we presented before could be generalized in the following way:

```
def comp[a, b, c] : (b ⇒ c) ⇒ (a ⇒ b) ⇒ a ⇒ c =
  f ⇒ g ⇒ x ⇒ f (g (x))
```

Now, instead of only composing operations that take integers and return integers, the function *comp* can compose any operation where the input type of the first operation is the same as the output type of the second operation.

Like methods, classes can themselves be parametrically polymorphic. For instance, to define a (homogeneous) list container, we could use a parametrized class to ensure that all elements are of the same type.

```
abstract class List[A]
case class Nil[A] extends List[A]
case class Cons[A] (x : A, xs : List[A]) extends List[A]
def len[A] (l : List[A]) : int = l match {
  case Nil ()        ⇒ 0
  case Cons (x, xs) ⇒ 1 + len (xs)
}
```

Bounded Polymorphism In Scala we can also have a form of *bounded* polymorphism by using *type parameter bounds*. This kind of polymorphism is useful in situations where we want to parameterize a method or a class by some type but we need to assume some operations on that type. A typical example where this kind of polymorphism is useful is when we want to define ordered insertion on lists.

$$\begin{array}{l}\textbf{def}\ \mathit{insert}[A <: \mathit{Ordered}[A]]\ (x : A, l : \mathit{List}[A]) : \mathit{List}[A] = l\ \textbf{match}\ \{\\ \quad \textbf{case}\ \mathit{Nil}\ () \quad\quad \Rightarrow \mathit{Cons}\ (x, \mathit{Nil}[A])\\ \quad \textbf{case}\ \mathit{Cons}\ (y, ys) \Rightarrow \textbf{if}\ (x \leqslant y)\ \mathit{Cons}\ (x, \mathit{Cons}\ (y, ys))\ \textbf{else}\ \mathit{Cons}\ (y, \mathit{insert}\ (x, ys))\\ \}\end{array}$$

In this situation we need to assume that the elements of type A contained in the list have the operation $<$. This is ensured by bounding the type A by $\mathit{Ordered}[A]$.

Variance Annotations Scala's generic types have, by default, non-variant subtyping. However, it is possible to annotate the type parameter to change its variance: if we prefix a formal type parameter with a '+' we change the subtyping so that it becomes covariant; alternatively, if we prefix a '−' we change the subtyping so that it becomes contravariant.

Functions In Scala, functions are objects (since all values are objects). The *Function1* trait is defined as follows.

$$\begin{array}{l}\textbf{trait}\ \mathit{Function1}[-a, +b]\ \{\\ \quad \textbf{def}\ \mathit{apply}\ (x : a) : b\\ \}\end{array}$$

This trait defines the interface of a function with one input type a and an output type b. Note that the input type is contravariant, while the output type is covariant. Besides *Function1*, there are also definitions for functions of many other arities: there is one definition for each possible number of function parameters. The syntax $(T1, ..., Tn) \Rightarrow S$ is just a Scala abbreviation for the parameterized type $\mathit{Functionn}[T1, ..., Tn, S]$.

2.1.5 Abstract Types

Scala has the notion of abstract types, which provide a flexible way to abstract over concrete types used inside a class/trait declaration. Abstract types are used to hide information about internals of a component, in a way similar to their use in SML (Harper and Lillibridge, 1994) and OCaml (Leroy,

1994). When creating a new object from a class that has abstract types, we need, as with all other members of a class, to initialize them with concrete types. Abstract types are considered by Odersky and Zenger (2005b) as essential for the construction of reusable components and they allow information hiding over several objects that, as argued in Section 1.2, is a key part of component-oriented programming. We start by discussing abstract types using a simple example first and we discuss a real-world example next, when we present *family polymorphism and self types*.

Consider a toy class *AbstractType* and one object instance *t*:

```
abstract class AbstractType {
  type A
  val x : A
}
def t = new AbstractType {type A = int; val x = 0; }
```

One interesting feature of abstract types is that they can act existentially or universally. The following definition, for example,

```
def func[a] (t : AbstractType) : a = t.x // type error
```

incurs on a type error because *t.x* has type *t.A*, but *t.A* is not on scope (much like an existential type). Alternatively, we could define

```
def func[a] (t : AbstractType {type A = a}) : a = t.x
```

which is a valid definition. The difference is that now *t.A* is in scope and is unified with *a*. This usage of abstract types is similar to parametric types. In fact, it is possible to model generics with abstract types (Odersky, 2006a).

Family polymorphism and self types We shall now see how to use abstract types to model families of types that vary together covariantly. This concept is known as *family polymorphism*. In particular we shall see how to capture the Observer pattern as a library of code. This example is taken from Odersky (2006a).

In the Observer pattern there are two kinds of participants: *subjects* and *observers*. The subjects define the methods *subscribe* and *publish*, which are used, respectively, to register observers and to notify all the observers. The notification of the observers is done by calling the method *notify* on the observer instances. A system that captures this design pattern is presented in Figure 2.1. The top-level class *SubjectObserver* contains two nested classes *Subject* and *Observer*, which correspond

```
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer
  abstract class Subject requires S {
    private var observers : List[O] = List ()
    def subscribe (obs : O) =
      observers = obs :: observers
    def publish =
      for (val obs ← observers) obs.notify (this)
  }
  trait Observer {
    def notify (sub : S) : unit
  }
}
```

Figure 2.1: The OBSERVER pattern in Scala

to the subject and observer participants of the pattern.

Note that the *Subject* and *Observer* do not refer directly to each other. Instead, the abstract types *S* and *O* are used to replace the "hard" references. It is this use of abstract types that allows the system to be extended covariantly. The use of **requires** in

```
abstract class Subject requires S {...
```

expresses that *Subject* can only be instantiated if its concrete class conforms to *S*. The type *S* is called a *self-type* of *Subject*. The use of self-types means that the type of **this** inside the class is the actual self-type (in the example, the type of **this** is *S*). The self-type annotation is needed in this example to ensure that the call *obj.notify* (**this**) is type-correct.

Subclasses of *SubjectObserver* can define application-specific subjects and observers. The example we use next is *SensorReader* that takes sensors as subjects and displays as observers. In this example, we instantiate the types *S* and *O* to *Sensor* and *Display*, which implement, respectively, *Subject* and *Observer*.

```
object SensorReader extends SubjectObserver {
  type S = Sensor
  type O = Display
  abstract class Sensor extends Subject {
    val label : String
    var value : double = 0.0
```

```
    def changeValue (v : double) = {
      value = v
      publish
    }
  }
  class Display extends Observer {
    def println (s : String) = System.out.println (s)
    def notify (sub : Sensor) =
      println (sub.label + " has value " + sub.value)
  }
}
```

This combination of abstract types, self-types and nested classes allows us to have information hiding over several objects, which is one of the premises of COP.

2.1.6 Implicit Parameters

Scala's *implicit parameters* allow some parameters to be inferred implicitly by the compiler (the inference process is guided by types) and can be used to emulate Haskell's type classes (Hall *et al.*, 1996) as noted by Odersky (2006b).

We shall see how this works by looking at another example presented in Odersky (2006a). Consider a definition of the concept of a *Monoid* in Scala. We could define an approximation using:

```
trait Monoid[a] {
  def unit : a
  def add (x : a, y : a) : a
}
```

The Haskell reader should notice the similarity between the monoid trait and the standard type class declaration for monoids. An example object would be a monoid on strings, with the unit being the empty string and addition being the concatenation of strings.

```
implicit object stringMonoid extends Monoid[String] {
  def unit : String = ""
  def add (x : String, y : String) : String = x.concat (y)
}
```

Again, the Haskell reader should note the connection between the implicit *stringMonoid* object and the instance declaration for string monoids in Haskell.

Ignoring the **implicit** keyword for a moment, we could start defining operations that are generic in the monoid. For example:

```
def sum[a] (xs : List[a]) (implicit m : Monoid[a]) : a =
  if (xs.isEmpty) m.unit
  else m.add (xs.head, sum (xs.tail) (m))
```

We can now use *sum* in the following way (as we would have done normally):

```
def test : String = sum (List ("a", "bc", "def")) (stringMonoid)
```

However, alternatively we could skip the second argument since the compiler has enough information to infer it automatically.

```
def test2 : String = sum (List ("a", "bc", "def"))
```

This works because the **implicit** keyword in the object informs the compiler that *stringMonoid* is the default value for the type *Monoid*[*String*]. The **implicit** keyword in the definition of *sum* informs the compiler that the argument *m* may be skipped if there exists one implicit object with the type of *Monoid* [*a*] in scope. When defining *test2* the compiler infers that the type of *m* should be *Monoid* [*String*] and since the programmer did not specify *m* it uses the implicit value *stringMonoid*.The second use of *sum*, with the implicit parameter inferred by the compiler, is similar to ad-hoc overloaded functions in Haskell (by using type class constraints).

As we have seen, implicit parameters give us a way to emulate Haskell's *type class* mechanism is Scala but, in some sense, implicit parameters are more flexible than type classes because we can choose to use something else other than the default value, which is not possible with Haskell's type classes.

2.1.7 Higher-kinded Types

Type constructor polymorphism and constructor (type) classes have proved themselves very useful in Haskell, allowing the definition of concepts such as monads (Wadler, 1993), applicative functors (McBride and Paterson, 2007) or many other container-like abstractions. This motivated the recent addition of type constructor polymorphism to Scala (Moors *et al.*, 2007), which allows similar constructions to be defined. For example, the most recent incarnation of the *Iterable* class is defined in Scala as:

```
trait Iterable[El, Container[_]] {
  def map[NewEl] (f : El ⇒ NewEl) : Container[NewEl]
  def flatMap[NewEl] (f : El ⇒ Iterable[NewEl]) : Container[NewEl]
  def filter (p : El Boolean) : Container[El]
}
```

The thing to note is that the *Iterable* is parametrized by *Container* [_], which is a type that is itself parametrized by another type. In other words, *Container* is a type constructor. Because we parametrize over the type constructor *Container* we can use it in the method definitions with any other types. In particular, in the definition of *map*, we can see that the return type of that method will be *Container*[*NewEl*], where *NewEl* is a type parameter of the method *map*. With parametrization by types only, it would not be possible to write this definition of *Iterable* and we would have to content ourselves with something less expressive.

2.2 Scala as a DGP language

Recalling that the three kinds of parametrization for a programming language to support DGP are parametrization by shape, type and computation, we can see that Scala readily supports the latter two. Parametrization by type is ensured by the generics support; and parametrization by computation is ensured by the support for higher-order functions. However, it is not so clear that Scala supports parametrization by shape. Moors *et al.* (2006) show how to encode *type-constructor polymorphism* and with that encoding develop a simple datatype-generic programming library. In this section we shall see how the type-constructor polymorphism encoding works and how it can be used to develop generic programming libraries. Our implementation differs from Moors *et al.*, but it is conceptually similar.

2.2.1 Encoding Type-Constructor Polymorphism

When we have a **trait** like:

```
trait TypeConstructor {
  type A
}
```

we can refer to either *TypeConstructor* {**type** $A = \ldots$} or to *TypeConstructor*. As discussed in Section 2.1.5, the former acts universally on the type *A*, while the latter acts existentially. Because

TypeConstructor is just a regular Scala type, we can abstract over type parameters of that kind. When we do so, we essentially get type-constructor polymorphism. For example, the notion of a functor, which defines an operation *fmap* on some type-constructor *F*, can be defined in Scala as:

```
trait Functor[F <: TypeConstructor] {
  def fmap[a, b] : (a ⇒ b) ⇒ F {type A = a} ⇒ F {type A = b}
}
```

Note that the type parameter of *Functor* is bounded to the type *TypeConstructor* and not to *TypeConstructor* {**type** $A = \ldots$}. If we could only use *TypeConstructor* universally we would be forced to have something like

```
trait BadFunctor[a, F <: TypeConstructor {type A = a}] {
  def fmap[a, b] : (a ⇒ b) ⇒ ? ⇒ ?
}
```

which would not allow us to have the right types for the second argument of *fmap* and for its output type. With the initial definition of *Functor*, however, we can abstract over *F* and still refine its type parameter *A* in the definition of *fmap*.

2.2.2 A Little DGP Library

Having shown how to encode type-constructor polymorphism and how to implement the notion of *Functor* in Scala, we now turn into implementing a small DGP library based on the Haskell library presented in Gibbons (2006).

The first step is to define a type *Mu* that is parametrized by a type constructor *F*. *Mu* may be seen as a template for constructing datatypes (in technical terms it is a type-level fixpoint combinator). The parameter *F* is the *shape* of the datatype — different shapes will give rise to different datatypes.

```
trait Mu[F <: TypeConstructor] {
  def In : F {type A = Mu[F]}
}
```

Defining Datatypes Lists are an example of a datatype that can be defined using *Mu*. First we define the shape of lists by creating a type constructor *ListF*. Then we define the two possible shapes for lists (a list either is empty, or has an element and another list) using the classes *NilF* and *ConsF*. Finally, we define a *Functor* object that defines the *fmap* operation for lists.

```
trait ListF extends TypeConstructor
case class NilF[a] extends ListF {type A = a}
```

```
case class ConsF[a] (x : Int, xs : a) extends ListF {type A = a}
implicit object FunctorList extends Functor[ListF] {
  def fmap[a, b] : (a ⇒ b) ⇒ ListF {type A = a} ⇒ ListF {type A = b} =
    f ⇒ {
      case NilF () ⇒ NilF ()
      case ConsF (x, xs) ⇒ ConsF (x, f (xs))
    }
}
```

Having defined the shape, we can define the datatype simply by applying *Mu* to the *ListF*.

```
type IntList = Mu[ListF]
```

The two constructors for lists can then be defined as follows.

```
def Nil : IntList = new IntList {def In = NilF[IntList]}
def Cons (x : Int, xs : IntList) : IntList = new IntList {def In = ConsF[IntList] (x, xs)}
```

Defining Generic Functions We can define operations that work for every instance of *Mu* (regardless of *F*). For example the *catamorphism* recursion pattern (Meijer *et al.*, 1991) can be defined once and for all as

```
def cata[F <: TypeConstructor, a] (f : F {type A = a} ⇒ a)
                                  (implicit ft : Functor[F]) : Mu[F] ⇒ a =
  comp (f) (comp (ft.fmap (cata[F, a] (f))) (.In))
```

and operations that follow those recursion patterns can be simply defined by instantiating the shape, type and functional parameters of the recursion pattern accordingly.

```
def sumList : IntList ⇒ Int =
  cata[ListF, Int] {case NilF () ⇒ 0; case ConsF (x, n) ⇒ x + n}
```

2.3 VISITORS and COMPOSITES

In this section we will show how the VISITOR and COMPOSITE patterns (Gamma *et al.*, 1995) are traditionally presented. The COMPOSITE pattern is closely related to the VISITOR since most implementations of visitors use it implicitly, which justifies a separate presentation.

2.3.1 The COMPOSITE Pattern

Unlike many functional programming languages, most object oriented languages do not have a built-in mechanism to define data types. Instead, the combination of composition and subclassing

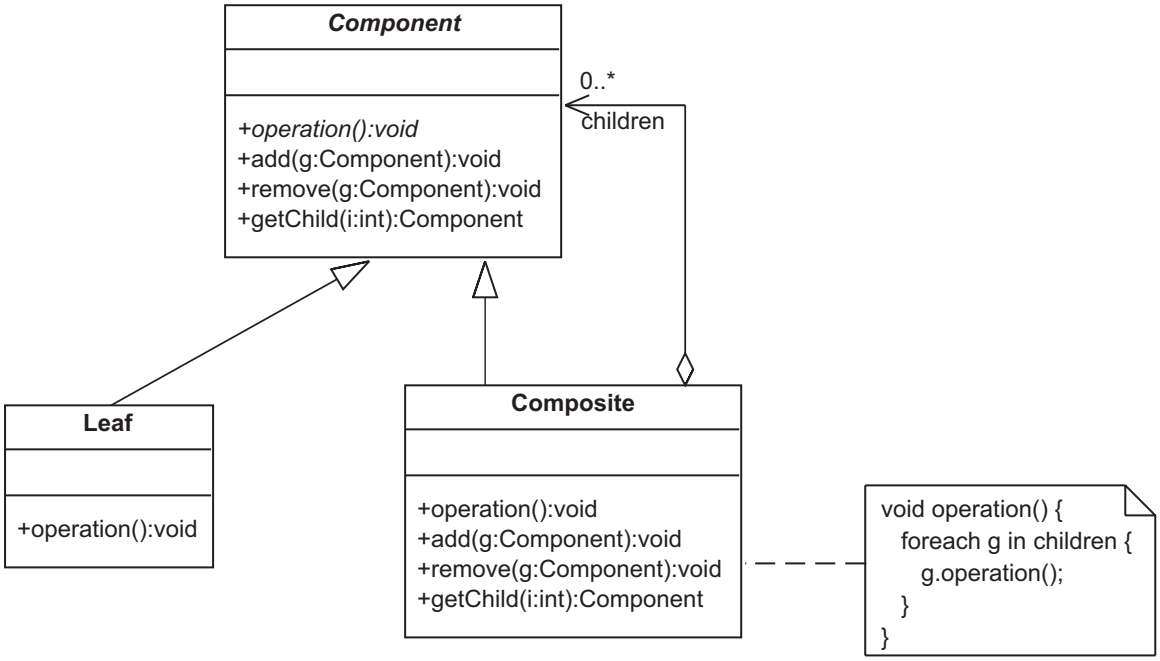


Figure 2.2: The Composite Design Pattern

can be used to define hierarchical structures. The aim of the COMPOSITE design pattern is to informally describe how to design such structures. The pattern consists of two types of components:

- The *Element* (or *Component*), usually represented as an interface, describes all the operations that can be applied to such structure.
- *Concrete Elements* are subtypes of *Element* and they are analogous to a value constructor of a datatype.

In Figure 2.3.1 we have a typical class diagram for the COMPOSITE showing how the components interact. The *Component* specifies all the operations allowed in an interface or abstract class. There are two kinds of concrete elements: *Leafs* and *Composites*. The former has no children, while the later can have children.

A difference between datatypes and COMPOSITES is that with COMPOSITES we need to define in advance all the operations that can be used in the structure. This is not the case with data types, where we can define functions at any point. In contrast, data types need to define all the variants at once while COMPOSITES can be extended — by adding new subclasses of *Element* — at any point.

The two forms of hierarchical structures (data types and COMPOSITES) can be used, most of the time, for solving the same problems. However, there are situations where one approach is preferable

to the other. This fact creates a tension that the programmer needs to solve: is it preferable to use data types and be able to extend the set of functions at any point; or, is it better to use the object oriented approach and be able to add variants at any point? There are even situations where it would be desirable that the two forms of extensibility co-exist. Wadler (1998) identified this issue and named it the *expression problem*.

It is possible, in an object-oriented setting, to have hierarchical structures that have the same form of extensibility as datatypes. The VISITOR design pattern, that we shall discuss in the next section, explains how this can be achieved.

2.3.2 The VISITOR Pattern

The VISITOR design pattern is an alternative way to implement structures that separates the operations from the object structure, thus allowing us to add new operations without changing the object structure. Moreover, the VISITOR keeps related aspects of a single operation together, by defining them in a single class. Figure 2.3 shows the UML class diagram for the design pattern. The components collaborate as follows:

- the *Visitor* interface declares a *visit* method for each *ConcreteElement* type;
- each *ConcreteVisitor* class implements a single operation, defining the *visit* method for each *ConcreteElement*;
- the *Element* abstract superclass (which is not actually required to make the implementation work) declares the *accept* method, taking a *Visitor* as argument;
- each *ConcreteElement* subclass defines the *accept* method to select the appropriate *visit* method from a *Visitor*.

The trade-off encapsulated by the VISITOR pattern is that while new operations are easy to add, adding new variants is hard; this is the opposite of the standard object-oriented approach. Therefore, the VISITOR is best applied to problems where the object structure rarely changes, which is a perspective is more akin to the functional programming paradigm, and can be considered as a functional idiom within an object-oriented paradigm. In particular, the VISITOR can be compared with *datatypes* and *pattern matching* in functional programming languages, where the hierarchical

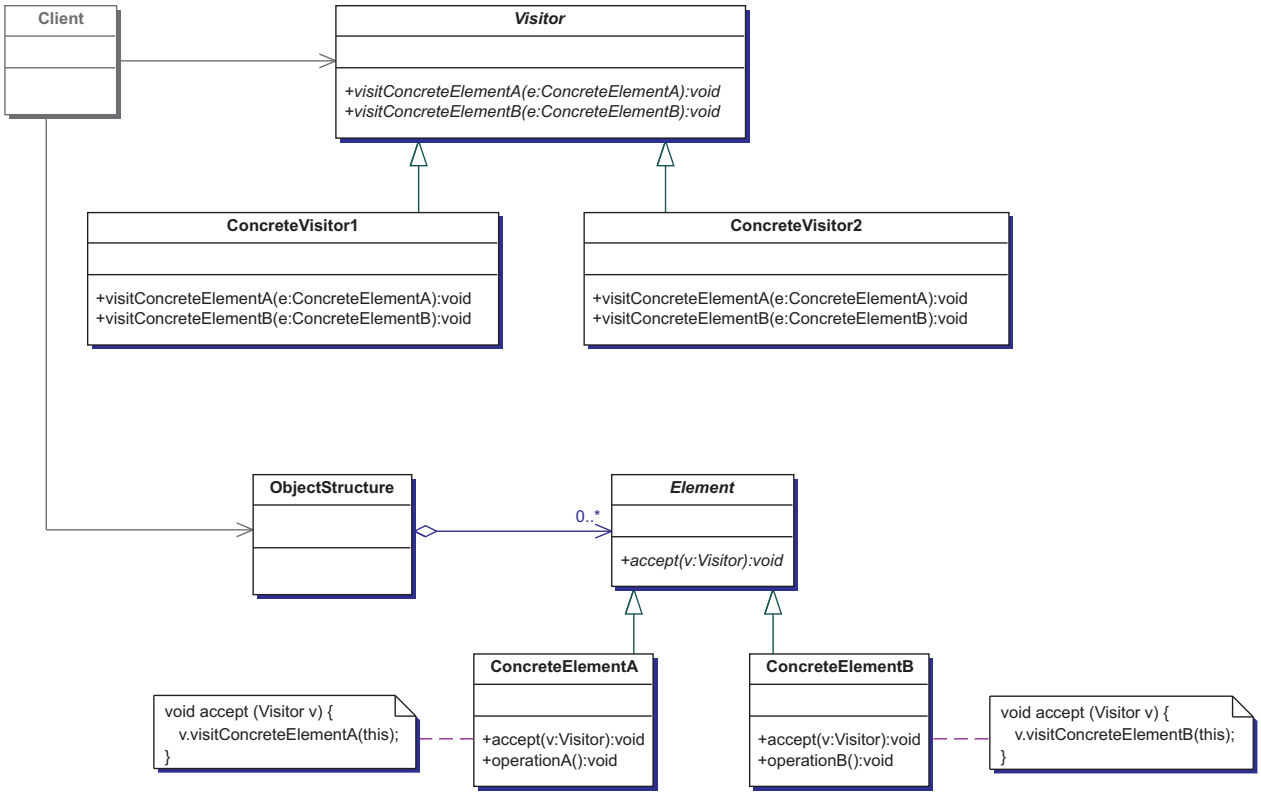


Figure 2.3: The VISITOR design pattern

structure (normally a COMPOSITE) defined by the *Element* superclass and *ConcreteElement* subclasses corresponds to the datatype, and the different *ConcreteVisitors* correspond to definitions via pattern matching.

Imperative and Functional VISITORS

In the original presentation of the visitor pattern, the *visit* and *accept* methods do not have a return value (or, more accurately, they return *void*). However, another possibility would be to have the *visit* and *accept* methods to return a value corresponding to some computation performed during the traversal. We shall make that distinction, using some terminology borrowed from Buchlovsky and Thielecke (2005):

Definition 1 (Imperative Visitor) An *imperative visitor* has *visit* and *accept* methods that return *void*; all computations are executed through side-effects, accumulating results via mutable state.

□

Definition 2 (Functional Visitor) A *functional visitor* is immutable; all computations return their results through the return values of the *visit* and *accept* methods, which are pure. □

Imperative visitors have two advantages over functional visitors. Firstly, functional visitors require some kind of generics, since visitors computing values of different types require *visit* and *accept* methods with different return types. In contrast, imperative visitors have the same interface, even when computing values of different types. Secondly, there is a minor performance penalty to be paid with functional visitors, due to the overheads imposed by propagating results via return values rather than directly via internal variables.

However, there are a few reasons why a functional presentation of the VISITOR may be preferable. The first reason concerns composability: because the *accept* method returns the result of the computation, we can immediately feed that value to another method without having to extract the state from the visitor class; that is, the result of the traversal can be obtained through an expression rather than a statement. Perhaps more importantly, it is easier to reason about functional visitors because computation is a visible effect and not an implicit side-effect of the methods: assuming no other side-effects, the *accept* methods can be seen as pure mathematical functions.

Example: Binary Trees Figure 2.4 shows how to implement binary trees using a functional VISITOR in Scala. The trait *Tree* and the classes *Empty* and *Fork* define a COMPOSITE where *Tree* is the *Element* and *Empty* and *Fork* are two *Concrete Elements*. The method *accept*, defined in *Tree* and implemented in the two concrete elements, takes a *TreeVisitor* object that has two *visit* methods (one for each concrete element). This whole system of classes defines an instance of the VISITOR pattern. Unlike with the traditional presentation of the VISITOR the parameters of the constructors are fed directly into the *visit* methods instead of passing the whole constructor object. Parametrizing the *visit* methods in this way gives, as we shall see, a very functional programming feel when programming with visitors.

In order to define new functions we need to create *ConcreteVisitor* objects. For example, a function to compute the *depth* of a binary tree could be defined as follows:

```
object Depth extends TreeVisitor[int] {
  def empty = 0
  def fork (x : int, l : Tree, r : Tree) =
    1 + max (l.accept (this), r.accept (this))
}
```

```
trait Tree {//The Element
  def accept[R] (v : TreeVisitor[R]) : R
}
case class Empty extends Tree {
  def accept[R] (v : TreeVisitor[R]) : R = v.empty
}
case class Fork (x : int, l : Tree, r : Tree) extends Tree {
  def accept[R] (v : TreeVisitor[R]) : R = v.fork (x, l, r)
}
trait TreeVisitor[R] {//The Visitor
  def empty : R
  def fork (x : int, l : Tree, r : Tree) : R
}
```

Figure 2.4: A Functional VISITOR for Binary Trees

When the *Tree* is *Empty*, then the case for empty trees (the *empty* method) returns 0. Otherwise, the tree is a branch (handled by the *fork* method), and we return one more than the maximum of the depth of the two subtrees. In order to compute the depth of the subtrees, we need to call the *accept* method of those trees with the *Depth* visitor — this is, in essence, a recursive call.

Defining values of type *Tree* benefits from Scala's **case class** syntax, because there is no need to have redundant uses of the **new** keyword. To use a *ConcreteVisitor*, we need to pass it as a parameter to the *accept* method of a *Tree* value. As a simple example, here is how to define a method *test* that computes the depth of a small tree.

```
def atree = Fork (3, Fork (4, Empty, Empty), Empty)
def test = atree.accept (Depth)
```

Relationship with Functional Programming To conclude this section, we compare functional visitors with datatypes and pattern matching in standard functional languages.

In functional programming, datatype declarations are commonly used to define structures. For example, we could define our binary trees as:

```
data Tree where
  Empty :: Tree
  Fork  :: Int → Tree → Tree → Tree
```

Comparing this with functional visitors we can see that, despite the extra code required by the visitor approach, there is a very close relationship between this datatype declaration and the

Element and *ConcreteElement* types. In particular, the constructors have the same parameters: *Empty* takes no parameters, so is a constant of type *Tree*, and *Fork* takes an integer value together with two subtrees and constructs a *Tree*. This relationship would not be so obvious had we used imperative visitors instead.

Functions can be defined by *pattern matching* over those values of data types. The *depth* function could be defined as:

$$\begin{aligned} &\mathit{depth} && :: \mathit{Tree} \rightarrow \mathit{Int} \\ &\mathit{depth}\ \mathit{Empty} && = 0 \\ &\mathit{depth}\ (\mathit{Fork}\ x\ l\ r) && = 1 + \mathit{max}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r) \end{aligned}$$

This definition bears remarkable similarities to the *Depth* visitor presented in the previous subsection. The *visit* methods correspond to the different cases of the definition by pattern matching, and the recursive calls take the place of the subtree calls to the *accept* method. Functional concrete visitors can be regarded as a very close relative to functions defined by pattern matching.

2.4 Functional Notation

Throughout this thesis we will use a functional notation to specify parts of our Scala code. This notation is, essentially, pretty printed Haskell code where Haskell's specific technical details are removed for clarity. This notation can also be interpreted as a fairly standard extension of System F (sometimes we assume that primitive types like *Int* exist or we assume that System F has built-in recursion). We shall use identifiers started with capital letters denote types (or type variables) and identifiers started with lower case to denote (value) variables. Function types are denoted as $A \Rightarrow B$, universal quantification is denoted $\forall A.$ **type** (where **type** is some valid type possibly defined using A). New definitions can be introduced by specifying a name, a type, and an expression:

$$\begin{aligned} &\mathit{mydef} \in \mathbf{type} \\ &\mathit{mydef} \equiv \mathbf{exp} \end{aligned}$$

Expressions consist of variables, applications and lambda abstractions. The application of a function f to an argument x is defined as $f\ x$ and lambda abstractions are of the following form $\lambda\mathbf{var} \Rightarrow \mathbf{exp}$.

We also allow the introduction of new types using the syntax $\mathit{mytype} \equiv \mathbf{type}$ where *mytype* is just a shorthand for **type**. Extra syntactic sugar includes lambda expressions ranging over multiple

variables ($\lambda var_1\ var_2\ \ldots\ var_n \Rightarrow$ **exp**); and the declaration of variables on the left-side of the definition $f\ x\ y \equiv$ **exp** (syntactic sugar for $f \equiv \lambda x \Rightarrow \lambda y \Rightarrow$ **exp**).

Whenever we feel the need we will provide the corresponding Haskell code in annexe.

Chapter 3

Visitors as Encodings of Datatypes

The VISITOR design pattern is related to algebraic data types, providing a functional programming style in an object-oriented setting. In this chapter, we explore this association and argue that variants of the pattern are related to known encodings of datatypes — in particular, Church and Parigot encodings. We use this relationship to capture the VISITOR pattern as a generic library, parametrized by the shape of the datatype and also by the decomposition strategy, while maintaining type safety. We also show how to implement a functional notation for visitors that allows the definition of functions (on those visitors) that resemble definitions by pattern matching in conventional functional languages.

3.1 Introduction

Many functional programming languages provide built-in support for algebraic datatypes via datatype declarations. As an example, consider a datatype declaration for peano naturals in a functional programming style.

data *Nat* **where**
  Zero :: *Nat*
  Succ :: *Nat* → *Nat*

The **data** declaration introduces a new type constructor and new value constructors. The type constructor *Nat* classifies values that are built with the corresponding value constructors *Zero* and

Succ.

An extra benefit of AlgDTs is that we get *pattern matching* for free, which allows us to exploit the shape of the datatype in order to define functions. For example,

$$\begin{aligned} &\mathit{toInt} && :: \mathit{Nat} \to \mathit{Int} \\ &\mathit{toInt}\ \mathit{Zero} && = 0 \\ &\mathit{toInt}\ (\mathit{Succ}\ n) && = 1 + \mathit{toInt}\ n \end{aligned}$$

defines the function that converts a peano natural into a built-in integer.

Even though many functional programming languages have built-in support for AlgDTs, it is possible to encode AlgDTs just using functions, thus not requiring special support. However, those encodings tend to require quite sophisticated type features, which are not available in many languages.

The best known encoding of datatypes is the *Church encoding* (Böhm and Berarducci, 1985). This encoding derives from System F, and allows us to write *iterative definitions* over the structure of the datatypes. However, there are certain functions that are inherently inefficient or even inexpressible using iteration alone. A less well-known encoding is the *Parigot encoding* (Parigot, 1992), which allows *generally recursive* definitions, but requires System F itself to be extended with recursion.

Object-oriented languages do not generally have native support for AlgDTs, and mostly rely on class hierarchies and composition to define aggregate structures. The Composite design pattern (Gamma *et al.*, 1995) provides a good model for OO programmers wanting to define recursive tree-structured aggregates. The Visitor pattern (Gamma *et al.*, 1995), allows the definition of operations on the elements of an object structure (typically defined using a Composite) without changing the classes of those elements on which it operates. In this chapter, we will discuss two variants of the Visitor: *internal* visitors, which control the traversal themselves, and *external* visitors, in which the traversal is controlled by the client.

Visitors correspond, in a very close sense, to encodings of data types. In particular, internal visitors are related to Church encodings and external visitors are related to Parigot encodings. We do not claim credit for this observation; it seems to be folklore knowledge (one way or another) among some communities. In fact, Buchlovsky and Thielecke (2005) present a type-theoretic formalization of the relation between encodings of datatypes and Visitors.

Our main goal is to use the existing theory behind these encodings to present the Visitor pattern

as a *software component* instead of the traditional informal design pattern presentation. Essential to this goal is the existence of programming languages like Scala, which provide standard OO constructs but also have powerful type systems capable of capturing such abstractions.

We start by discussing a design choice that needs to be made before any implementation of the VISITOR pattern in Section 3.2. The original contributions of the Chapter follow:

- Section 3.3 builds on Buchlovsky and Thielecke's observation that one variant of the VISITOR pattern corresponds to Church encodings, showing how to model the corresponding visitors in Scala.

- Section 3.4 extends the observation, looking at Parigot encodings and describing how they lead to a different variation of the VISITOR pattern, which again we capture in Scala.

- Section 3.5 presents two generalizations of the above encodings. The first, *datatype-genericity*, is well-known in the domain of type theory; it allows parametrization by the shape of data being traversed. The second, which we call *strategy-genericity*, is novel as far as we know; it allows parametrization by the decomposition strategy, with instantiations to (among others) internal and external strategies for controlling the traversal.

- Section 3.6 uses these generalizations to build a highly generic library of visitors in Scala. The library supports datatype-genericity and strategy-genericity, allowing the programmer to avoid an early commitment in either of these dimensions. All this is achieved without sacrificing type safety.

- Section 3.7 shows how we can have a more intuitive notation for visitors, implemented directly in Scala, that allows, for example, visitors to be treated as functions that take composites (datatypes) as parameters.

- Section 3.8 explores the expressiveness of the visitor library and shows that the library is capable of expressing *parametric*, *mutually-recursive* and *existential* visitors. Furthermore, using the connection between recursion patterns and the decomposition strategy, and inspired by Meertens (1992) work on *paramorphisms*, we show how to encode *paramorphic visitors*.

Finally, a discussion of the results and some related work is presented in Section 3.9.

```
trait Tree {//The Element
  def accept[R] (v : TreeVisitor[R]) : R
}
case class Empty extends Tree {
  def accept[R] (v : TreeVisitor[R]) : R = v.empty
}
case class Fork (x : int, l : Tree, r : Tree) extends Tree {
  def accept[R] (v : TreeVisitor[R]) : R =
    v.fork (x, l.accept (v), r.accept (v))
}
trait TreeVisitor[R] {//The Visitor
  def empty : R
  def fork (x : int, l : R, r : R) : R
}
```

Figure 3.1: A functional internal VISITOR for binary trees.

3.2 Internal or External VISITORS: A Design Choice

In the GoF presentation of the design pattern, the question of who is responsible for traversing the object structure is raised. In the example in Section 2.3.2 we decided that the responsibility should belong to the *visit* methods of the concrete visitors. However, alternatively, we could put that responsibility on the *accept* methods of the concrete elements. Figure 3.1 shows such alternative implementation of a visitor. The difference between the two alternatives shows up for recursive occurrences of *Tree*. The concrete element *Fork* has two recursive occurrences of *Tree*, and its *accept* method passes on its *Visitor* parameter to each of these trees before calling the *visit* method. This contrasts with the approach taken in the Section 2.3.2, whereby the two subtrees were passed, unchanged, as arguments to the *visit* method. Consequently, the *visit* method will have the occurrences of the type *Tree* replaced by *R*. Following Buchlovsky and Thielecke (2005), we introduce the following terminology.

Definition 3 (Internal Visitor) An *internal visitor* is an instance of the VISITOR pattern in which the responsibility for traversing the hierarchical structure is assigned to the *accept* methods of the concrete element classes. □

Definition 4 (External Visitor) An *external visitor* is an instance of the VISITOR pattern in which the responsibility for traversing the hierarchical structure is assigned to the *visit* methods of the concrete visitor classes. □

There are trade-offs between the two options: internal visitors are simpler to use and have more interesting algebraic properties, but the fixed pattern of computation makes them less expressive than external visitors.

3.3 Internal Visitors and the Church Encoding

In this section, we look at the well-known *Church encoding* of datatypes in the lambda calculus (Böhm and Berarducci, 1985), and see that it is directly related to internal visitors.

3.3.1 Encoding Data Types in the Lambda Calculus

In the pure lambda calculus, there is no native notion of datatype. Nonetheless, data types can be encoded just using functions. Church himself observed that natural numbers can be encoded in the untyped lambda calculus via what is now known as the *Church numerals*, in which numbers are encoded by repeated function composition: the number 0 is represented by 'zero-fold composition', the number 1 by 'one-fold composition', the number 2 by 'two-fold composition', and so on.

$$\begin{aligned} &\mathit{zero} \equiv \lambda f\ x \Rightarrow x \\ &\mathit{succ} \equiv \lambda n \Rightarrow f\ x \Rightarrow f\ (n\ f\ x) \end{aligned}$$

The functions *zero* and *succ* can be used to construct numerals. For example, in order to represent 1 and 2 we would use:

$$\begin{aligned} &\mathit{one} \equiv \mathit{succ}\ \mathit{zero} \\ &\mathit{two} \equiv \mathit{succ}\ \mathit{one} \end{aligned}$$

Standard mathematical operations like addition, multiplication and exponentiation can be defined as:

$$\begin{aligned} &m + n \equiv \lambda f\ x \Rightarrow m\ f\ (n\ f\ x) \\ &m \times n \equiv \lambda f\ x \Rightarrow m\ (n\ f)\ x \\ &m \uparrow n \equiv \lambda f\ x \Rightarrow n\ m\ f\ x \end{aligned}$$

Numerals are not the only data structures that can be encoded using this technique; the technique generalizes to many other datatypes. For example, the constructors for the binary trees that we presented in Figure 3.1 can be encoded in the untyped lambda calculus as follows:

$$\begin{aligned} &\mathit{empty} \equiv \lambda e\ f \Rightarrow e \\ &\mathit{fork} \quad \equiv \lambda x\ l \Rightarrow r\ e\ f \Rightarrow f\ x\ (l\ e\ f)\ (r\ e\ f) \end{aligned}$$

The Church encoding brings significant extra insights when we move from an untyped to a typed lambda calculus such as System F as it reveals deep connections with (intuitionistic second order) logic arising from the *Curry-Howard correspondence* (Howard, 1980). In a minor variant of System F — with native support for integers — the Church numerals and our binary trees of integers can be given the following types:

$$Nat \equiv \forall X.\ (X \Rightarrow X) \Rightarrow X \Rightarrow X$$
$$Tree \equiv \forall X.\ X \Rightarrow (Int \Rightarrow X \Rightarrow X \Rightarrow X) \Rightarrow X$$

3.3.2 The Church Encoding in Scala

Noting that classes are just record types and that records themselves can be thought of being tuples of named components, we can transform the functional encodings that we have just presented into their uncurried form and using two classes, we can encode those types. To demonstrate, consider our type for natural numbers $Nat \equiv \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X$ for which the isomorphic uncurried type is to $\forall X.((X \Rightarrow X) \times X) \Rightarrow X$. By defining $NatAlgebra\ X \equiv (X \Rightarrow X) \times X$ and encoding *NatAlgebra* as a trait in Scala, we obtain:

```
trait NatAlgebra[X] {
  def succ (n : X) : X
  def zero : X
}
```

The type $Nat \equiv \forall X.NatAlgebra\ X \Rightarrow X$ can itself be encoded in Scala with the following trait:

```
trait Nat {
  def accept[X] (alg : NatAlgebra[X]) : X
}
```

The universal quantification of X appearing in the definition of *Nat* is translated into a *generic method* over type variable X. Using Visitor terminology, the trait *NatAlgebra* [X] is the *Visitor* type of the element superclass *Nat*.

Our Visitor for peano numerals is internal, since the two constructors of *Nat*, which are the concrete elements, are defined as:

```
case class Zero extends Nat {
  def accept[X] (alg : NatAlgebra[X]) : X ≡ alg.zero
}
case class Succ (n : Nat) extends Nat {
  def accept[X] (alg : NatAlgebra[X]) : X ≡ alg.succ (n.accept (alg))}
```

As we can see, it is the responsibility of *accept* to iterate through the recursive parts of the structure by calling *accept* recursively in the *Succ* case.

A similar translation process could be applied to the example of binary trees, resulting in the code already presented in Figure 3.1 (except that what would be *TreeAlgebra* here is called *TreeVisitor* there).

3.4 External Visitors and the Parigot Encoding

In the previous section we related the *Church encoding* of datatypes with an instance of the VISITOR design pattern. In particular, this encoding leads to *internal visitors*. In this section we will look at another, less well-known, encoding of datatypes, and see how it leads to *external visitors*.

3.4.1 Limitations of Church Encodings

The *Church encoding* of datatypes is the most well-known encoding of datatypes; however, it has some limitations. In particular, some definitions cannot be written efficiently (or written at all) in this style. A well known example is the predecessor function on naturals, which takes linear time to compute with the Church encoding. Informally, the reason for the limitation is that the visitor does not control the recursive calls; rather, they are automatically called in the *accept* method. To demonstrate this limitation consider a function that tests whether or not a tree is empty. Using an internal visitor, we could define:

```
def isEmpty = new TreeVisitor[boolean] {
  def empty = true
  def fork (x : int, l : boolean, r : boolean) = false
}
```

While this function would work, it would take linear time to do so: the recursive calls are automatically made in the *accept* method, regardless of whether their results are used (at least, without making use of lazy evaluation).

$$
\begin{array}{ll}
\mathit{Nat} & \equiv \forall A.(\mathit{Nat} \Rightarrow A) \Rightarrow A \Rightarrow A \\
\mathit{zero} & \in \mathit{Nat} \\
\mathit{zero} & \equiv \lambda s\ z \Rightarrow z \\
\mathit{succ} & \in \mathit{Nat} \Rightarrow \mathit{Nat} \\
\mathit{succ}\ n & \equiv \lambda s\ z \Rightarrow s\ n \\
\mathit{Tree} & \equiv \forall A.A \Rightarrow (\mathit{Int} \Rightarrow \mathit{Tree} \Rightarrow \mathit{Tree} \Rightarrow A) \Rightarrow A \\
\mathit{empty} & \in \mathit{Tree} \\
\mathit{empty} & \equiv \lambda e\ f \Rightarrow e \\
\mathit{fork} & \in \mathit{Int} \Rightarrow \mathit{Tree} \Rightarrow \mathit{Tree} \Rightarrow \mathit{Tree} \\
\mathit{fork}\ x\ l\ r & \equiv \lambda e\ f \Rightarrow f\ x\ l\ r
\end{array}
$$

Figure 3.2: Parigot encodings of naturals and binary trees.

3.4.2 Parigot Encodings in the Lambda Calculus

The *Parigot encoding* (Parigot, 1992) is another way to encode datatypes. Parigot encodings require a version of System F extended with recursion. This extension allows us to express recursive (as opposed to iterative) definitions.

We show Parigot encodings of the naturals and our binary trees in Figure 3.2. Note that, unlike with the Church encodings, the types *Nat* and *Tree* occur recursively in their own definitions. Also, the definitions of constructors with recursive occurrences (the *succ* and the *fork* constructors) are simpler than those in the Church encoding, because we can feed the recursive parameter directly into the handling functions s and f.

3.4.3 The Parigot Encoding in Scala

When we translate the Parigot encoding of our binary trees using the same approach as we did for the Church encoding, we obtain the code presented in Figure 2.4. With this visitor, we can define a function that computes whether a tree is empty or not as follows:

```
def isEmpty = new TreeVisitor[boolean] {
  def empty = true
  def fork (x : int, l : Tree, r : Tree) = false
}
```

While this definition looks much like the function in Section 3.4.2 (the only visible syntactical differences are the types of l and r), the resulting behaviour is significantly different: instead of processing all the elements of the *Tree* structure and taking linear time, as the Church encoding

does, this version takes only constant time to compute, since l and r are not recursively processed.

3.5 Generic Visitors: Two Dimensions of Parametrization

We shall see in this section that the encodings presented previously can be generalized such that, with a single construction, we have a *datatype-generic* form of the encoding. This allows us to precisely capture the notions of internal and external visitors. Still, we are confronted with two incompatible notions of visitors that force the programmer to a design choice along with its advantages and disadvantages. We will show that this need not be the case by introducing a second dimension of genericity that we shall refer to as *strategy-genericity*. With strategy-genericity the choice between internal and external visitors is parametrizable, and a single common definition can be provided. This will allow us to define, in Section 3.6, a highly generic library for Visitors. The Haskell code corresponding to the functional specification presented in this section can be found in Appendix A.

3.5.1 Abstracting over the shape

Church encodings and, to a lesser extent, Parigot encodings have been well studied in the domain of type theory. A particularly relevant theoretical result is the fact that those encodings can be characterized generically. Each of these characterizations abstracts from the differences between visitors for different shapes of data structure, allowing for definitions that are generic in this shape. The generic template for defining datatype-generic versions of both Church and Parigot encodings is of the form $\forall X.\ (F\ R \Rightarrow X) \Rightarrow X$, with Church encodings becoming $\mathit{Church}\ F \equiv \forall X.\ (F\ X \Rightarrow X) \Rightarrow X$ and Parigot encodings becoming $\mathit{Parigot}\ F \equiv \forall X.\ (F\ (\mathit{Parigot}\ F) \Rightarrow X) \Rightarrow X$. Normally, F is required to be a *Functor* (i.e. F allows a mapping operation) and it is common to use sums and products functors to encode a variety of (polynomial) datatypes. Because those results are well established, we shall not delve into details. For the interested reader, we refer to Böhm and Berarducci (1985); Parigot (1992); Buchlovsky and Thielecke (2005).

Visitors as Products of Functions

The traditional presentation of encodings of datatypes in System F (and common variants) is of the form $T \equiv \forall X.\ (F\ R \Rightarrow X) \Rightarrow X$. In this form a datatype T can be defined by instantiating $F\ R$ to

some sum-of-product functor $\sum_i F_i\ R$. Buchlovsky and Thielecke (2005) show that a variation of these encodings, of the form $T \equiv \forall X.\ (\prod_i F_i\ R \Rightarrow X) \Rightarrow X$, can be precisely related to the Visitor pattern. We can easily show that the two encodings are, in reality, isomorphic using the laws of exponentials:

$$
\begin{aligned}
&(\prod_i F_i\ R \Rightarrow X) \Rightarrow X \\
\Leftrightarrow\ &\textit{exponential} \\
&(\prod_i X^{F_i\ R}) \Rightarrow X \\
\Leftrightarrow\ &\textit{product of exponentials} \\
&(X^{\sum_i F_i\ R}) \Rightarrow X \\
\Leftrightarrow\ &\textit{exponential} \\
&((\sum_i F_i\ R) \Rightarrow X) \Rightarrow X
\end{aligned}
$$

With Buchlovsky and Thielecke's variation a new datatype T can be defined by providing a product of functions $V\ R\ X \equiv \prod_i F_i\ R \Rightarrow X$ (the visitor), where each function $F_i\ R \Rightarrow X$ corresponds to a *visit* method and $F_i\ R$ corresponds to the arguments of the constructor. It is easy to see that when $V\ R\ X \equiv F\ R \Rightarrow X$ we would obtain the traditional form of encodings. Church and Parigot encodings (corresponding, respectively, to internal and external visitors) follow from two specific instantiations of R.

Definition 5 (Generic Internal Visitors)

Let $V\ R\ X$ be a product of functions of the form $\prod_i F_i\ R \Rightarrow X$. Then, for $R \equiv X$, we can define *generic internal visitors* as:

$$\textit{Internal}\ V \equiv \forall X.V\ X\ X \Rightarrow X$$

□

Definition 6 (Generic External Visitors)

Let $V\ R\ X$ be a product of functions of the form $\prod_i F_i\ R \Rightarrow X$. Then, for $R \equiv \textit{External}\ V$, we can define *generic external visitors* as:

$$\textit{External}\ V \equiv \forall X.V\ (\textit{External}\ V)\ X \Rightarrow X$$

□

The motivation for this generalization stems from the fact that the visitor components of the Visitor pattern are, in essence, products of functions — each *visit* method is a function and the

$$NatF\ r\ a \equiv (a, r \Rightarrow a)$$
$$Nat \equiv Internal\ NatF$$
$$zero \in Nat$$
$$zero \equiv \lambda(z, s) \Rightarrow z$$
$$succ \in Nat \Rightarrow Nat$$
$$succ\ n \equiv \lambda(z, s) \Rightarrow s\ (n\ (z, s))$$

Figure 3.3: Church encoding of Peano numerals using products of functions

whole class is, therefore, representing a product of functions. In the two definitions above, V is a type parameter that is abstracting over concrete visitor components. It can be said that V is the *shape parameter* of the encodings (since different instantiations of V will lead to different datatypes).

3.5.2 Abstracting over the decomposition strategy

By using generic encodings based on products of functions it is possible to abstract from differences in the shape of data and model different decomposition strategies — internal and external — of visitors that are generic in the shape. Still, there is substantial duplication of code whenever we want to have both strategies. However, this duplication is not necessary because, as we shall see, we can model visitors that are generic in both the shape and the strategy.

The template $\mu\ V \equiv \forall X.\ V\ R\ X \Rightarrow X$ can be used, as we have seen, to capture different implementations of the Visitor pattern. However, $\mu\ V$ cannot be captured linguistically because R is unbound. Therefore, we need to replace R with something that is linguistically valid if we want to have a truly *strategy* generic visitor component (this is, a component that can be parametrized by its implementation strategy). Since R represents the type of recursive occurrences that appear in the visit methods we can see that, if we want to capture both internal and external visitors, R should depend on both V and X. We can make this dependency explicit by making $R \equiv S\ V\ X$ and bind S universally.

$$\mu\ V \equiv \forall S\ X.\ V\ (S\ V\ X)\ X \Rightarrow X$$

We shall refer to S as the *decomposition strategy* (or just *strategy*).

Although $\mu\ V$ is linguistically capturable it is still not right. To see what the problem is, lets first reformulate the Church peano numerals using products of functions, as in Figure 3.3. Now lets see what happens when we try to use $\mu\ NatF$ instead of *Internal NatF*:

$$Nat \equiv NatF$$
$$zero \in Nat$$
$$zero \equiv \lambda(z, s) \Rightarrow z$$
$$succ \in Nat \Rightarrow Nat$$
$$succ\ n \equiv \lambda(z, s) \Rightarrow s\ ?$$

As we can see, we have no problems defining the contructor *zero*. However the story is different for *succ*: it is impossible to provide a value of the right type for *s* since *s* requires an argument with type $S\ V\ X$ and we cannot create any values of that type because S is universally quantified. The solution for this problem consists in adding some extra information about S in the definition of μ.

$$\mu\ V \equiv \forall X\ S.\ Decompose\ S \Rightarrow V\ (S\ V\ X)\ X \Rightarrow X$$

The extra information is given by $Decompose\ S$. In essence $Decompose\ S$ is basically just a type-overloaded (in the type-parameter S) method. In other words, the implementation of this method can be determined solely from the type S and, therefore, we can make $Decompose\ S$ implicit. Referring to the method in $Decompose\ S$ as dec_S we have that:

$$dec_S \in V\ (S\ V\ X)\ X \Rightarrow \mu\ V \Rightarrow S\ V\ X$$

The operation dec_S solves the problem of producing a value of type $S\ V\ X$ and allow us to define the constructor *succ* as:

$$succ \in Nat \Rightarrow Nat$$
$$succ\ n \equiv \lambda(z, s) \Rightarrow s\ (dec_S\ (z, s)\ n)$$

(Note that the parameter $Decompose\ S$ is implicitly passed). In order to define new strategies we need to define some concrete type S and the corresponding dec_S operation. For example, to make internal and external visitors two instances of $\mu\ V$ we specialize S to *Internal* and *External*:

$$Internal\ V\ X \equiv X$$
$$External\ V\ X \equiv \mu\ V$$

(Here we reuse the identifiers *Internal* and *External* to refer to the associated decomposition strategies.) The specific instantiations of *dec* for internal and external visitors are:

$$dec_{Internal} \in (V\ (Internal\ V\ X)\ X) \Rightarrow \mu\ V \Rightarrow Internal\ V\ X$$
$$dec_{Internal}\ v\ c \equiv c\ v$$
$$dec_{External} \in (V\ (External\ V\ X)\ X) \Rightarrow \mu\ V \Rightarrow External\ V\ X$$
$$dec_{External}\ v\ c \equiv c$$

In the definition of $dec_{Internal}$ the reader should (again) note that the $Decompose\ S$ parameter is

implicitly passed and, therefore, the composite *c* just needs to take as an argument the visitor *v*. With $dec_{External}$, we simply ignore the visitor parameter and return the composite itself. This will then allow the programmer to use the composite directly in the definitions of the *visit* methods.

3.6 The VISITOR Pattern as a Library

In the previous section, we used the Church and Parigot encodings of datatypes to motivate a notion of visitors generic in two dimensions: in the shape of the data structure being visited, and in the strategy for assigning the responsibility of traversal. Armed with this insight, we will now present an implementation in Scala of a generic visitor library.

3.6.1 Defining the Library

The functional specification in the previous section can be translated into Scala without major issues (although the typing of the *Visitor* component is slightly different in the Scala version). The type μ that we defined previously corresponds to the *Composite*. We recall that definition (renaming μ to *Composite*) and annotate it extra information identifying the *accept* method and the visitor component.

$$Composite\ V \equiv \overbrace{\forall X\ S.\ Decompose\ S \Rightarrow \underbrace{V\ (S\ V\ X)\ X}_{Visitor} \Rightarrow X}^{accept\ \ method}$$

Visitors

The *Visitor* component in the library, which corresponds to V in the functional specification, is parametrized by a strategy S and a result type X. The *Visitor* also contains a type R that corresponds to the type $S\ V\ X$ (the first argument of V, which specifies the type of recursive arguments).

```
trait Visitor {
  type X
  type S <: Strategy
  type R[v <: Visitor] = S {type X = Visitor.this.X; type V = v}
}
```

Here the reader may wonder if the type R could not have been typed as

type $R = S$ {**type** $X = Visitor$.**this**.X; **type** $V >:$ $Visitor$.**this**.**type** }

The intention being that *Visitor*.**this**.**type** would refer to the type of the concrete subclass of Visitor in use. However there are issues with this solution as this type is not precise enough. Although there would be other Scala solutions that would capture a type for *Visitor* that is similar in expressiveness to the functional specification, we opted to parametrize *R* with a visitor. This solution gives us some extra flexibility over the functional specification while allowing us to give precise typings to our visitor. This small inconsistency with the functional specification is discussed in more detail in Section 6.3.1.

Composites

The *Composite* trait is parametrized by a visitor *V* and contains an *accept* method that takes two parameters. The first parameter is the visitor to apply; the second is the decomposition strategy to use while visiting the structure.

```
trait Composite[−v <: Visitor] {
  def accept[s <: Strategy, x] (vis : v {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x
}
```

Strategies

The decomposition strategy is encoded in Scala with the following trait :

```
trait Strategy {
  type V <: Visitor
  type X
  type Y
  def get : Y
}
```

A *Strategy* is parametrized by a visitor *V*, type *X* and a type *Y* that will be defined in terms of *V* and *X*. Subtypes of this trait will correspond to different possible decomposition strategies for the visitors. In particular, the strategies *Internal* and *External* are defined as:

```
trait Internal extends Strategy {
  type Y = X
}
trait External extends Strategy {
  type Y = Composite[V]
}
```

The method *get* retrieves the result of the recursion (whose type is given by *Y*).

The decomposition strategy parameter in the *accept* method can be made implicit. This means that we can call the *accept* method passing just the first parameter, if there is in scope one decomposition operation of the appropriate *Decompose* type for the second argument. The trait *Decompose* is parametrized by the decomposition strategy *S* and it encapsulates a single method *dec*. This method takes a visitor and a composite and returns the result of recurring on that composite using the decomposition strategy.

```
trait Decompose[s <: Strategy] {
  def dec[v <: Visitor, x]
      (vis : v {type X = x; type S = s}, comp : Composite[v]) : s {type V = v; type X = x}
}
```

Decomposition strategies for internal and externals visitors are provided by the library (note that both strategies can be used implicitly):

```
implicit def internal : Decompose[Internal] = new Decompose[Internal] {
  def dec[v <: Visitor, x]
      (vis : v {type X = x; type S = Internal}, comp : Composite[v]) =
    new Internal {type V = v; type X = x; def get = comp.accept (vis)}
}
implicit def external : Decompose[External] = new Decompose[External] {
  def dec[v <: Visitor, x]
      (vis : v {type X = x; type S = External}, comp : Composite[v]) =
    new External {type V = v; type X = x; def get = comp}
}
```

The two implementations of the method *dec* correspond, respectively, to the definitions $dec_{Internal}$ and $dec_{External}$ in the functional specification. The important thing here — effectively the piece of code that we want to abstract from — is the definition of *get*, which is *comp.accept* (*vis*) for internal visitors and just *comp* for external visitors. In other words, the decomposition strategy of the internal visitors recurs on the composite *comp* (since it calls the *accept* method); and the decomposition strategy for external visitors returns the composite untouched, which allows concrete visitors to control recursion themselves.

```
type Tag = String
type Attrs = List[Pair[String, String]]
trait XMLVisitor extends Visitor {
  type Rec = R[XMLVisitor]
  def text   (s : String) : X
  def entity (x : Tag, l : Attrs, r : List[Rec]) : X
}
type XML = Composite[XMLVisitor]
case class Text (x : String) extends XML {
  def accept[s <: Strategy, x] (vis : XMLVisitor {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.text (x)
}
case class Entity (x : Tag, l : Attrs, r : List[XML]) extends XML {
  def accept[s <: Strategy, x] (vis : XMLVisitor {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.entity (x, l, mapList ((x : XML) ⇒ decompose.dec[XMLVisitor, x] (vis, x), r))
}
```

Figure 3.4: A simplified form of XML documents as visitors.

3.6.2 Using the Library

We shall now show how to use this library to define visitors, and how the different kinds of visitors can be used simply by parametrizing them accordingly. As a motivating example, we will define a simplified form of XML documents, and provide functions to convert them into strings and to test the documents for equality.

A Simple XML Module

Figure 3.4 shows the visitor and composite components for a form of simplified XML documents. A document is either some text or an entity; an entity has a tag, a list of attributes and a list of XML documents as children. The trait *XML* is the composite, with two case classes *Text* and *Entity* being the two concrete elements of the composite; the trait *XMLVisitor* is the visitor.

Printing XML Documents

Figure 3.5 shows a function *printXML* that transforms an XML document into a string. The function is defined using an internal *XMLVisitor*. The internal strategy is configured by setting the type

```
def printXML (x : XML) : String =
  x.accept[Internal, String] (new XMLVisitor {
    type X = String
    type S = Internal
    def printAttrs (l : Attrs) : String =
      l match {
        case Nil ⇒ ""
        case Pair (s1, s2) :: xs ⇒ "  " + s1 + "=\"" + s2 + "\"" + printAttrs (xs)
      }
    def printListXML (x : List[Rec]) : String =
      x match {
        case Nil ⇒ ""
        case y :: ys ⇒ y.get + "\n" + printListXML (ys)
      }
    def text (s : String) = s
    def entity (x : Tag, l : Attrs, r : List[Rec]) =
      "<" + x + printAttrs (l) + ">\n" + printListXML (r) + "<" + x + "/>"
  })
```

Figure 3.5: A printing function for XML documents

parameter *S* and its associated value parameter *dec* accordingly. The return type *X* is set to *String*. The function is then defined by implementing the two methods *text* and *entity* in the *XMLVisitor*. In the *text* case, we simply return that text. In the *entity* case, we produce the corresponding XML symbols while printing the attributes (using an auxiliary function *printAttrs* whose definition we omit) and the list of XML documents. Because we use an internal visitor, the elements of the list passed to the method *printListXML* are the strings resulting from recursive processing of the children; these subresults are combined by interspersing line breaks.

Comparing XML Documents

We now present an example of a binary method, checking whether two XML documents are equal. Binary methods are notoriously difficult to define in a type-safe way in most object-oriented languages, but they are not a problem if we use external visitors. The basic idea, presented in Figure 3.6, is to use nested case analysis. We start by calling the *accept* method for the first document *x*, using an external visitor that returns a boolean. If the visitor discovers that *x* is a text, it decomposes *y* with another external visitor that analyses its shape: if *y* is also a text it compares the two strings, otherwise it returns false. Dually, when *x* is an entity, another visitor is used to analyse *y*;

```
def equalsXML (x : XML, y : XML) : boolean =
  x.accept[External, boolean] (new XMLVisitor {
    // Setting up the parameters of the visitor
    type X = boolean
    type S = External
    // Defining the cases
    def text (s1 : String) = y.accept[External, boolean] (new XMLVisitor {
      type X = boolean
      type S = External
      def text (s2 : String) : boolean = s1.equals (s2)
      def entity (x : Tag, l : Attrs, r : List[Rec]) : boolean = false
    })
    def entity (x1 : Tag, l1 : Attrs, r1 : List[Rec]) : boolean =
      y.accept[External, boolean] (new XMLVisitor {
        type X = boolean
        type S = External
        def equalsList[a] (f : a ⇒ a ⇒ boolean, x : List[a], y : List[a]) : boolean =
          Pair (x, y) match {
            case (Pair (Nil, Nil))        ⇒ true
            case (Pair (x :: xs, y :: ys)) ⇒ f (x) (y) ∧ equalsList (f, xs, ys)
            case (Pair (xs, ys))          ⇒ false
          }
        def text (s2 : String) = false
        def entity (x2 : Tag, l2 : Attrs, r2 : List[Rec]) : boolean =
          x1.equals (x2) ∧ l1.equals (l2) ∧
            equalsList ((x : Rec) ⇒ (y : Rec) ⇒ equalsXML (x.get, y.get), r1, r2)
      })
  })
```

Figure 3.6: An equality function for XML documents using visitors.

this visitor returns false if *y* is a text and returns the result of comparing the tags, attributes and children when *y* is an entity. Note that because we use an external visitor, we need to manually apply *equalsXML* recursively to all the XML occurrences.

3.7 Syntactic Sugar for VISITORS in Scala

One criticism that may be made of our visitor library is that it is somewhat verbose to use. In this section, we shall see how we can improve our notation for visitors using some advanced features of the Scala programming language. With this notation, the use of our library becomes more intuitive

and less cumbersome.

3.7.1 Extending the Library

Functional notation

The invocation *a.accept* (*f*) where *a* is a composite and *f* is a visitor can be interpreted as a form of reverse application *f* (*a*), where *f* plays the role of a function and *a* is the argument of that function. This observation can be quite elegantly expressed in Scala by making *Visitors* instances of functions; this is possible in Scala because *all functions are objects*. The *Function1* class from the Scala standard library is the class of all functions with one argument (Odersky, 2006a).

trait *Function1*[−*S*, +*T*] {**def** *apply* (*x* : *S*) : *T*}

We can make our visitors instances of *Function1*, but not directly, because we would need to have the abstract type members of *Visitor* in scope for defining the inheritance relation. The (somewhat obscure) solution is to create a class *VisitorFunc* whose subclasses are also *subtypes* of *Visitor*. However *VisitorFunc* (unlike *Visitor*) uses type arguments instead of the abstract type members. This allows us to use those type parameters in the inheritance relation. We also need to create *VFunction1* (a subclass of *Function1*) that has an extra field with a decomposition strategy. The code is presented next:

```
abstract class VFunction1[s <: Strategy, a, b] (dec : Decompose[s])
  extends Function1[a, b] {def decompose = dec}
abstract class VisitorFunc[v <: Visitor, s <: Strategy, x] (dec : Decompose[s])
  requires (VFunction1[s, Composite[v], x] with v {type X = x; type S = s})
  extends VFunction1[s, Composite[v], x] (dec) {
    type X = x
    type S = s
    def apply (c : Composite[v]) : x = c.accept[s, x] (this) (decompose)
  }
```

The difficult thing is getting the inheritance relation right. In order for a visitor to become a function it must be a subclass of *VFunction1* and, therefore, *VisitorFunc* needs to extend that class. Concrete visitors *v* will then use mixin composition to combine themselves with *VisitorFunc*. The *apply* method coming from *Function1* will take a *Composite* as first argument, and that composite needs to know which particular visitor *v* we need; although we know we are defining a visitor, we still do not know which particular visitor is required by the composite. Fortunately, Scala's self-types

allow us to require a type for the self-reference **this** that can depend on information that will only be known at a later stage. In our case this information concerns the parametrized concrete visitor *v* that will be known when we define the subclasses of *VisitorFunc*.

Shortcuts for *VisitorFunc*

If we want to make use of the functional notation, we additionally need to define a new class that mixes in the concrete visitor with *VisitorFunc*. For example, for our XML example we could have:

abstract class *VXML*[*s* <: *Strategy*, *b*] (**implicit** *decompose* : *Decompose*[*s*])
  extends *VisitorFunc*[*XMLVisitor*, *s*, *b*] (*decompose*)
  with *XMLVisitor*

This class also has the advantage of being more concise to use than the visitor directly (because we use generic parameters instead of abstract types).

Automatic coercions

Finally, for convenience, we provide automatic coercions between the *Strategy* objects and the values that are contained in the *get* field — effectively, this means that the *get* method is automatically called whenever a value of that type is required.

implicit def *internal2a*[*x*] (*x* : *Internal* {**type** *X* = *x*}) : *x* = *x.get*
implicit def *external2mu*[*v* <: *Visitor*, *x*]
  (*x* : *External* {**type** *V* = *v*; **type** *X* = *x*}) : *Composite*[*v*] = *x.get*

3.7.2 Using the Extended Library

It is now time to see how the notation helps defining functions that make use of visitors. We shall demonstrate by re-writing the equality function presented in Section 3.6 with the notation from this section. Figure 3.7 shows the *equalsXML* function with the new notation (note that we skip the definition of *equalsList*). In particular, we can see two of the three notational extensions: instead of using *XMLVisitor* we now use *VXML* thus setting all the relevant parameters for the use of external visitors with XML documents. The other thing to notice is the absence of *get* methods, since we use automatic coercions to convert between *Rec* and *XML*.

The functional notation becomes handy when we want to have named visitors instead of anonymous visitors. For example, suppose that we decided to implement *printXML* with an external visitor. Normally we would have something like

```
def equalsXML (x : XML, y : XML) : boolean =
  x.accept[External, boolean] (new VXML[External, boolean] {
    def text (s1 : String) = y.accept[External, boolean] (new VXML[External, boolean] {
      def text (s2 : String) : boolean = s1.equals (s2)
      def entity (x : Tag, l : Attrs, r : List[Rec]) = false
    })
    def entity (x1 : Tag, l1 : Attrs, r1 : List[Rec]) =
      y.accept[External, boolean] (new VXML[External, boolean] {
        def text (s2 : String) = false
        def entity (x2 : Tag, l2 : Attrs, r2 : List[Rec]) =
          x1.equals (x2) ∧ l1.equals (l2) ∧
            equalsList ((x : Rec) ⇒ (y : Rec) ⇒ equalsXML (x, y), r1, r2)
      })
  })
```

Figure 3.7: Equality re-written with the new notation

```
def printXML : VXML[External, String] = new VXML[External, String] {
    // code before omitted
  def printListXML (x : List[Rec]) : String =
    x match {
      case Nil ⇒ ""
      case y :: ys ⇒ y.get.accept (this) + "\n" + printListXML (ys)
    }
    // code before omitted
}
```

and the use of the recursive call would have to refer to the *accept* method. However, because *VXML* is a subclass of *Function1* we can make use of the functional notation, with the resulting code looking like:

```
def printXML : VXML[External, String] = new VXML[External, String] {
    // code before omitted
  def printListXML (x : List[Rec]) : String =
    x match {
      case Nil ⇒ ""
      case y :: ys ⇒ printXML (y) + "\n" + printListXML (ys)
    }
    // code after omitted
}
```

Another consequence of this notation is that the clients of *printXML* can just use the functional notation in their calls. For example

def *testPrint* = *printXML* (*Entity* ("Name", *Nil*, *Text* ("ola") :: *Nil*))

can be written without calling the *accept* method on the *XML* document.

3.7.3 Comparison with Functional Programming

It is interesting to compare the Scala programs that we have developed in this and the previous sections with the equivalent program in a functional programming language. Using algebraic datatypes we could have a definition like (here we use Haskell's syntax):

data *XML* **where**
  Text :: *String* → *XML*
  Entity :: *Tag* → *Attrs* → *XML*

This definition corresponds, roughly, to the code presented in Figure 3.4. The definition of *XML* in Haskell is obviously much more elegant and intuitive than the corresponding Scala code. In Scala, case classes come close to this elegance without the need for a special purpose datatype declaration. One advantage of our visitors, however, is that they can be parametrized on their decomposition strategy. In other words, usual definitions of datatypes always came with a fixed decomposition strategy (which is normally equivalent to external visitors); with our visitors, we can choose different strategies. In order to obtain the same effect of, for example, internal visitors with conventional datatypes, we would need to write new functions (the *fold* combinators) for each datatype capturing that kind of traversal.

The code corresponding to the definition of *equalXML* (in Figure 3.7) could be written in the following way in a functional language:

```
equalsXML :: XML → XML → boolean
equalsXML x y =
  case x of
    (Text s1)          → case y of
      (Text s2)          → s1 ≡ s2
      (Entity x l r)     → False
    (Entity x1 l1 r1) → case y of
      (Text s2)          → False
      (Entity x2 l2 r2) → (x1 ≡ x2) ∧
        (l1 ≡ l2) ∧ equalsList (λx y → equalsXML x y) r1 r2
```

This definition, as we can see, is still more elegant than the one in Figure 3.7 using visitors. However, the syntactical disparity is much less here than it was with the datatype declaration. In

fact, defining functions using our visitor library is fairly practical even without any support from the compiler. Therefore, our library can be used in practice to define visitors and we do not have to worry about design decisions involving the control of the traversal (since this is parametrizable).

Our visitor library could be used to provide the semantics for possible AlgDTs language extensions. It would be interesting to explore a language extension that allowed us to explore strategy parametrization.

3.8 Expressiveness of the Visitor Library

In this section we will explore the expressiveness of our visitor library and see how it can be used to encode a large family of datatypes that includes *parametric*, *mutually recursive* and *existential* datatypes. The translation of datatypes is detailed, more formally, in Appendix B. Furthermore, using the connection between the decomposition strategy and recursion patterns, we will show that the internal and external visitors are not the only two kinds of visitors that can be encoded with our library. We will demonstrate this by presenting a visitor inspired by *paramorphisms* (Meertens, 1992).

3.8.1 Parametric Datatypes

Our visitor library can be used to encode parametric datatypes. In Figure 3.8 we show how parametric lists can be encoded in Scala using our visitor library. There are two constructors *Nil* and *Cons* with their corresponding visit methods (*nil* and *cons*) in the visitor component $ListVisitor[a]$. Worth noting is that $ListVisitor[a]$ is parametrized with a generic type a, which is the type of the elements in the list. The composite component is defined using a parametrized type synonym $List[a]$ where, again, the type a represents the types of the elements of the list. The class *VList* is provided as a convenience and allows us to use the functional notation provided by *VisitorFunc*. Two examples of functions defined over lists follow:

```
def sizeList[a] = new VList[Internal, a, int] {
  def nil              = 0
  def cons (x : a, xs : Rec) = 1 + xs
}
def addList = new VList[Internal, int, int] {
  def nil              = 0
```

```
trait ListVisitor[a] extends Visitor {
  type Rec = R[ListVisitor[a]]
  def nil : X
  def cons (x : a, xs : Rec) : X
}
type List[a] = Composite[ListVisitor[a]]
def Nil[a] : List[a] = new List[a] {
  def accept[s <: Strategy, x] (vis : ListVisitor[a] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.nil
}
def Cons[a] (x : a, xs : List[a]) : List[a] = new List[a] {
  def accept[s <: Strategy, x] (vis : ListVisitor[a] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.cons (x, decompose.dec[ListVisitor[a], x] (vis, xs))
}
abstract class VList[s <: Strategy, a, b] (implicit decompose : Decompose[s])
  extends VisitorFunc[ListVisitor[a], s, b] (decompose)
  with ListVisitor[a]
```

Figure 3.8: Parametric lists using the visitor library

```
  def cons (x : int, xs : Rec) = x + xs
}
```

The function *sizeList*, which computes the size of a list, shows how we can define a parametrically polymorphic function over our lists. The function *addList*, which computes the sum of all the elements of an integer list, shows how we can define a function over a list containing elements of a particular type.

3.8.2 Mutually Recursive Datatypes

With our library it is possible to define mutually-recursive visitors in a convenient way. The code in Figure 3.10 shows how we can encode trees in terms of forests, and forests in terms of trees using two mutually recursive visitors. Trees, whose type is given by *Tree*[*a*], have one constructor *Fork* that builds a tree containing one element and a forest. The trait *TreeVisitor*[*a*] is the corresponding visitor and has a visit methods *fork* matching the *Fork* constructor. Forests, whose type is given by *Forest*[*a*], have two constructors *Nil* and *Cons* that construct, respectively, an empty forest and a forest with one tree and another forest. The trait *ForestVisitor*[*a*] is the visitor component and the

```
def sumTree : VTree[Internal, int, int] = new VTree[Internal, int, int] {
  def fv = sumForest
  def fork (x : int, xs : R[ForestVisitor[int]]) = x + xs
}
def sumForest : VForest[Internal, int, int] = new VForest[Internal, int, int] {
  def tv = sumTree
  def nil                                                  = 0
  def cons (x : R[TreeVisitor[int]], xs : R[ForestVisitor[int]]) = x + xs
}
```

Figure 3.9: Adding elements in forests and trees of integers.

methods *nil* and *cons* match the *Nil* and *Cons* constructors. The fields *fv* and *tv* provide references to *TreeVisitor*[*a*] and *ForestVisitor*[*a*] that will be used when defining functions. For example, the two functions presented in Figure 3.9 show how we can define functions that add all the elements of a tree and a forest of integers. The definitions are mutually-recursive: *sumTree* uses *sumForest* in its definition and vice versa, by setting the *fv* and *tv* fields in the visitors.

Note that this formulation of mutually-recursive visitors is only possible because *R* is parametrized by a visitor. With the current functional specification, we cannot define mutually-recursive visitors in this way (although it is still possible to define them in different ways) because *R* is hard-wired to the concrete visitor being defined. It is worthwhile to the reader to see the discussions in Sections 3.6 and 6.3.1 and take a look at Appendix I to see how we can use a different functional specification setting that allows *R* to be parametrized by a visitor.

3.8.3 Existentially Quantified Datatypes

We can also define existentially quantified visitors with our library. In Figure 3.11 we show how to encode a form of heterogeneous lists (this is, lists that contain elements of different types) where the elements have an associated printing operation f. Like parametric lists we have two constructors *Nil* and *Cons*[*a*] that construct values of type *HList* and have corresponding visit methods *nil* and *cons*[*a*] in the trait *HListVisitor*. The existential types are achieved by universally quantifying the element type *a* at the visit methods (in this case we do that at the method *cons*). The trait *VHList* provides the functional notation. The function *printHList*, which uses the printing operation to print the values contained in the list, is defined as:

```
trait TreeVisitor[a] extends Visitor {
  def fv : ForestVisitor[a] {type X = TreeVisitor.this.X; type S = TreeVisitor.this.S}
  def fork (x : a, xs : R[ForestVisitor[a]]) : X
}
trait ForestVisitor[a] extends Visitor {
  def tv : TreeVisitor[a] {type X = ForestVisitor.this.X; type S = ForestVisitor.this.S}
  def nil : X
  def cons (x : R[TreeVisitor[a]], xs : R[ForestVisitor[a]]) : X
}
abstract class VTree[s <: Strategy, a, b] (implicit decompose : Decompose[s])
  extends VisitorFunc[TreeVisitor[a], s, b] (decompose)
  with TreeVisitor[a]
abstract class VForest[s <: Strategy, a, b] (implicit decompose : Decompose[s])
  extends VisitorFunc[ForestVisitor[a], s, b] (decompose)
  with ForestVisitor[a]
type Tree[a] = Composite[TreeVisitor[a]]
type Forest[a] = Composite[ForestVisitor[a]]
def Nil[a] : Forest[a] = new Forest[a] {
  def accept[s <: Strategy, x] (vis : ForestVisitor[a] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.nil
}
def Fork[a] (x : a, xs : Forest[a]) : Tree[a] = new Tree[a] {
  def accept[s <: Strategy, x] (vis : TreeVisitor[a] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.fork (x, decompose.dec (vis.fv, xs))
}
def Cons[a] (x : Tree[a], xs : Forest[a]) : Forest[a] = new Forest[a] {
  def accept[s <: Strategy, x] (vis : ForestVisitor[a] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.cons (decompose.dec (vis.tv, x), decompose.dec (vis, xs))
}
```

Figure 3.10: Visitors for the mutually-recursive *Forest* and *Tree* types.

```
trait HListVisitor extends Visitor {
  type Rec = R[HListVisitor]
  def nil : X
  def cons[a] (x : a, f : a ⇒ String, xs : Rec) : X
}
type HList = Composite[HListVisitor]
def Nil : HList = new HList {
  def accept[s <: Strategy, x] (vis : HListVisitor {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.nil
}
def Cons[a] (x : a, f : a ⇒ String, xs : HList) : HList = new HList {
  def accept[s <: Strategy, x] (vis : HListVisitor {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.cons (x, f, decompose.dec[HListVisitor, x] (vis, xs))
}
abstract class VHList[s <: Strategy, b] (implicit decompose : Decompose[s])
  extends VisitorFunc[HListVisitor, s, b] (decompose)
  with HListVisitor
```

Figure 3.11: Defining heterogeneous list with the visitor library

```
def printHList : HList ⇒ String = new VHList[Internal, String] {
  def nil                                = ""
  def cons[a] (x : a, f : a ⇒ String, xs : Rec) = "\n" + f (x) + xs.get}
```

3.8.4 Paramorphic Visitors

Internal and external visitors are not the only two possible decomposition strategies for visitors. Inspired by the connection between visitors and recursion patterns, we will now show how we can specify what we shall call *paramorphic* visitors — named after the *paramorphism* recursion pattern (Meertens, 1992). As mentioned in Section 3.5.2, the first thing we need to define is the strategy parameter:

$$Para\ V\ X = (X, Mu\ V)$$

The paramorphic decomposition strategy consists of a pair where the first component is the result of a recursive call and the second component is the substructure on which that call was made. It is interesting to observe that we could have equivalently written

$$Para\ V\ X = (Internal\ V\ X, External\ V\ X)$$

```
trait Para extends Strategy {
      type Y = Pair[X, Composite[V]]
}
implicit def external : Decompose[Para] = new Decompose[Para] {
   def dec[v <: Visitor, x] (vis : v {type X = x; type S = Para}, comp : Composite[v]) =
      new Para {
         type V = v; type X = x; def get = Pair[x, Composite[v]] (comp.accept (vis), comp)}
}
implicit def para2pair[v <: Visitor, x]
   (x : Para {type V >: v; type X = x}) : Pair[x, Composite[v]] = x.get
```

Figure 3.12: Paramorphic visitors in Scala.

so in a sense *Internal* and *External* are primitive entities and *Para* is not.

The operation dec_{Para} is defined as:

$$dec_{Para} :: (V\ (Para\ V\ X)\ X) \rightarrow Mu\ V \rightarrow Para\ V\ X$$

$$dec_{Para}\ t\ u = (u\ t, u)$$

In Appendix C the full Haskell code for the functional specification along with some examples is presented.

Paramorphic Visitors in Scala

In Scala, the necessary code for adding this new kind of visitor is presented in Figure 3.12. A new decomposition strategy entails the definition of three things:

1. The first thing that we need is a new strategy. In this case, the trait *Para* provides this strategy;
2. We also need to provide an Decompose object. For paramorphisms the operation *para* defines this;
3. Finally, for convenience, we provide one implicit coercion between instances of *Para* and the *Pair* provided by the field *get*. This coercion is given by *para2pair*.

After the library writer adds a new kind of visitor to the library, users can start defining their own functions. For example, the factorial function can be elegantly defined using paramorphic visitors as:

```
def fact : Nat ⇒ Nat = new VNat[Para, Nat] {
  def zero          = Succ (Zero)
  def succ (n : Rec) = mult (fst (n)) (Succ (snd (n)))
}
```

The benefit of using a paramorphic visitor for defining *fact* is that the definition *succ* (which matches the successor of a natural number) has access to both the result of the recursive call *fact* (*n*) (the first component of the pair) and *n* (the second component of the pair), making *fact* very easy to define. The full code necessary to run this example is presented in Appendix D.

3.9 Discussion

Design patterns are often described as "*elements of reusable object-oriented software*" (Gamma *et al.*, 1995). It is, perhaps, an irony that what is meant by "reusable" here is an extra-linguistic reuse in the form of "*prose, pictures and prototypes*" (Gibbons, 2003). This will necessarily manifest itself in programs through copy, paste and adapt practices. That is not to say that design patterns are a bad thing, but rather that they are not providing their maximum benefit. It would be better if "reusable" here were to have its traditional meaning in computing: the solutions provided by design patterns should be abstracted into modular pieces of software (or, in other words, components), which can be instantiated and applied without further change. This cannot be achieved in the mainstream languages of today such as Java and C#; but as we have shown, the more advanced features provided by more recent languages such as Scala do suffice. As (Gibbons, 2006, 2003) and others (Norvig, 1996; Arnout, 2004) have argued elsewhere, this issue of expressivity is "*not inherent in the patterns themselves, but evidence of a lack of expressivity in the languages of today*".

We have argued that the Visitor pattern is related to well-known encodings of data types from the type theory community. Building on the insights provided by this relation, we have shown that it is possible to capture the Visitor pattern as a modular software library, given a language with a sufficiently powerful type system. In particular we have shown that this can be achieved, *today*, in the Scala programming language. In addition to Scala's combination of functional and object-oriented features, the crucial feature that we need (and which is lacking in current mainstream languages) is the ability to parametrize on type constructors (which is closely related to *datatype-genericity*).

An interesting aspect of our library is that, not only does it allow us to capture a particular kind

of visitor (such as internal, or external) datatype-generically, but it also allows us to parametrize on the kind of the visitor itself. This increases even more the reusability of the pattern library, and strengthens the argument that design patterns can be captured linguistically. Rather than having to implement each variation separately, duplicating the common code, the alternatives arise as instantiations of a common abstraction. We believe that this also brings something new in more theoretical terms, since although it has been known that the template $(F\ U \Rightarrow A) \Rightarrow A$ is used by both the Church and Parigot encodings, we are not aware of any work that studied the two encodings as specializations of a more (linguistically capturable) general abstraction. In particular, the extraction of the commonalities via the *dec* operation seems to be novel.

Notions of *generic visitor* have been proposed in the past. Palsberg and Jay (1998) presented a solution relying on a reflection mechanism, where a single Java class *Walkabout* could support all visitors as subclasses. Refinements to the idea of using reflection to capture generic visitors, mostly to improve performance, have been proposed since (Grothoff, 2003; Forax *et al.*, 2005). Our work in this chapter differs from solutions based on reflection in two aspects. Firstly, one of the main motivations of these reflection-based generic visitors was to remove the need for an *accept* method in the hierarchical structure, since this is considered by many to be intrusive and against the object-oriented model. We do not share this motivation, since we interpret the VISITOR as an encoding of data types. Secondly, our solution is type-safe: we never get *"message not understood"* run-time errors. This is not the case for reflection-based approaches.

Another interesting feature of our library is the fact that we can use a *functional notation* for our visitors by interpreting the *accept* method as a form of reverse application. A similar idea motivates the *Peripaton* language, which supports the so-called "*visitor-oriented programming style*" (VanDrunen and Palsberg, 2004). In Peripaton, everything is a visitor: the visitor object can be considered the top of the object hierarchy, playing the same role as *Object* in Java. By interpreting the *visit* method as function application, we get a notion that lies in between functions and objects. We believe that this analogy with functions is a useful one and it lends a more intuitive notation to visitors. However, unlike VanDrunen and Palsberg, we do not force every object into a visitor/function and we do not require a language with special support for visitors. Instead, our interpretation is that visitors correspond to a subset of functions defined by (some kind of) pattern matching.

Chapter 4

Visitor-Generic Programming

In Chapter 3 we have seen that the Visitor pattern can be interpreted as a generic encoding of datatypes. This leads to a generic notion of a visitor that can be captured as a software library. With this library we can easily define functions for specific visitors; however, the library offers little help if we want to define functions that work for *any* visitors. Datatype generic programming (DGP) aims to solve this problem by allowing us to define *generic* functions that work for any shape (or visitor). In this chapter we develop a DGP library for visitors inspired by Hinze's GM approach. With this approach we can define our own generic functions on visitors. We also show that the GM implementation is itself an instance of the visitor pattern. However, it cannot be modelled directly with the current library because the family of visitors that GM belongs to is *type-indexed* (and the current library does not support that family). We show that it is possible to generalize our original visitor library to support *type-indexed* visitors. This insight allow us to eliminate the need for a design choice that is present in GM. Finally, we will see how to express a family of sums of products within our visitor library. Using this family we can express a wider range of generic functions.

4.1 Introduction

Suppose that we have datatypes of lists and trees of integers and that we want to add up the integers contained in values of those datatypes. Assuming that we defined visitors for those datatypes using

our library, it is easy to write two functions that do the job.

```
def addList = new VList[Internal, int] {
  def nil                  = 0
  def cons (x : int, xs : Rec) = x + xs
}
def addTree = new VTree[Internal, int] {
  def empty                      = 0
  def fork (x : int, l : Rec, r : Rec) = x + l + r
}
```

Since our library automatically provides decomposition strategies, we can just choose the most suitable strategy for solving our problem and define the two functions in a very simple way without worrying about the recursive boilerplate. For example, in the definitions above, we opted to use internal visitors to define the two addition functions for lists and trees, since the recursion pattern that is involved in these functions is a simple kind of structural recursion.

While our library can be very helpful when defining functions for particular datatypes, it offers little help if we want to define functions that work for *any* datatypes. If we now introduce a new kind of tree of integers and we wanted to add up all the integers of that tree, we would have to define yet another function that suits that particular datatype. A better option would be to define a generic function that works *once and for all* for all visitors.

DGP aims precisely at solving this problem. In this chapter we will show that we can extend our visitor library to support generic functions. In order to do so, we will translate the GM approach to DGP (Hinze, 2004) to Scala and show how to define generic functions on visitors. This will not require any modifications in our visitor library — although it will require that the users of the library construct their visitors based on sums of products, or establish an isomorphism between their visitor and sums of products.

GM is a particularly interesting lightweight approach that shows how to use Haskell's type classes to model a generic programming library based on sums of products. As it turns out, Hinze's own inspiration for GM (Hinze, 2006) comes from the same encodings of datatypes that we have been using in this thesis: he used Church and Parigot encodings to encode representations of datatypes in two different ways — this is no coincidence; the work presented in this thesis has been initially inspired by Hinze's work on lightweight approaches to DGP. These two alternative representations of datatypes give rise to two alternative implementations of DGP, which have different trade-offs and force the same design choice that we have with the original presentation of the

Visitor pattern.

In Section 4.2 we show how to encode sums and products in Scala, providing us with the basic machinery that we will need for supporting DGP. The contributions of this chapter follow:

- In Section 4.3 we provide a translation of GM into Scala, explain how we can define generic functions in this setting, and show two forms of reuse of generic functions. The first form of reuse, quite natural in an OO setting, is using inheritance. The second form of reuse is the so-called *local redefinition* (Löh, 2004), which can be used to override the behaviour of generic functions for parametric datatypes.

- In Section 4.4 we argue that the implementation of GM is itself an instance of the Visitor pattern, but it cannot be encoded using our visitor library. The reason is that GM requires an *indexed visitor*, which is out of reach in our initial library. We modify the visitor library to support indexed visitors, and show how an alternative implementation of GM could be constructed using this modified library. An immediate consequence of this alternative design is that the design choice between two alternative implementations of GM is unnecessary.

- In Section 4.5 we show how to we can can *view* another family of visitors as a particular instance of our visitor library and show that the GM approach readily supports a form of *views* (Holdermans *et al.*, 2006). In particular, we define a family of visitors based on sums of products. Furthermore, the use of this family will have some advantages in terms of performance, usability, expressiveness of generic functions compared with products of functions.

- In Section 4.6 we will develop a simple serialization library — a common application of generic programming. Moreover, we will show that, while the library can be defined for product-of-functions visitors, if we use the family of visitors defined in Section 4.5 we can use generic functions that express recursion patterns directly.

Finally, a discussion of the results and some related work is presented in Section 4.7.

4.2 Encoding Sums and Products in Scala

In this section we will show how to encode sums and products in Scala. This will be the basis for adding datatype-generic programming to our visitor library.

```
trait Plus[A, B] {
  def accept[t] (vis : PlusVisitor[A, B, t]) : t
}
case class Inl[A, B] (value : A) extends Plus[A, B] {
  def accept[t] (vis : PlusVisitor[A, B, t]) = vis.inl (value)
}
case class Inr[A, B] (value : B) extends Plus[A, B] {
  def accept[t] (vis : PlusVisitor[A, B, t]) = vis.inr (value)
}
trait PlusVisitor[A, B, T] {
  def inl (x : A) : T
  def inr (x : B) : T
}
```

Figure 4.1: A visitor for sums.

Sums Scala, like most object-oriented languages, does not contain a primitive notion of sums. Most of the time, a hierarchical design such as the Composite is used whenever different types of objects of a common kind are required. The Visitor pattern that we have employed in the previous chapters for implementing recursive data types can also encode sums. In fact, we could just use our library to define sums (and products). However, we shall refrain from doing that here because sums and products are non-recursive types; we opt for a more lightweight solution where we just apply the design pattern directly, which has the advantage of clarity. In Figure 4.1, we see an encoding of sums in Scala. The trait *Plus* has two type parameters A and B — which are respectively the types of the values of the two choices in the sum — and the standard *accept* method. There are two instances (*Inl* and *Inr*) of *Plus*, which are the constructors for the injections into the sum. Finally, the trait *PlusVisitor* is the visitor component and contains a method for handling each of the injections.

Products Products are easier to model in object-oriented languages. To model a product we just need a record with two fields and two type parameters (one field and one type parameter for each of the components of the product). We can also make products visitable by adding an *accept* method and defining the corresponding visitor (the trait *ProdVisitor*). The code for products as visitors is shown in Figure 4.2.

```
case class Prod[A, B] (fst : A, snd : B) {
  def accept[t] (vis : ProdVisitor[A, B, t]) : t = vis.prod (fst, snd)
}
trait ProdVisitor[A, B, T] {
  def prod (x : A, y : B) : T
}
```

Figure 4.2: A visitor for products.

Empty Product The final piece of machinery is the empty product, which has a single value. In other words, this is the neutral element of products and can be easily defined as:

```
trait One
```

Having defined the necessary machinery for sums of products, we can now move on to implement support for DGP.

4.3 Generic Programming with VISITORS

In the previous section we have shown how to encode sums and products in Scala. In this section, we will show how to use sums of products to define generic functions in Scala. This is achieved by applying the GM technique proposed by Hinze (2004) to Scala. We will also discuss two distinct mechanisms that can be used in Scala to reuse generic functions: *reuse by inheritance* and *local redefinition*.

4.3.1 Generics for the Masses in Scala

The technique presented by Hinze in GM shows how to encode generic functions within Haskell 98. In that proposal, a generic function can be encoded as an instance of a type class *Generic*. Another type class (the *Rep* class) defines a function *rep* that can be used to construct type representations automatically. As we shall see in Section 4.4, there is a relationship between Hinze's GM encoding and our work on the visitor pattern — the class *Generic* can be seen as the visitor component and the class *Rep* as the composite component. GM comes in two different flavours, which provide two slightly different interpretations of generic functions. The inspiration for these two different flavours comes from encodings of datatypes (specifically, the Church and Parigot encodings), which

were greatly explored in Chapter 3.

GM allows the definition of generic functions on datatypes that are isomorphic to sums of products. However, these isomorphisms need to be explicitly defined, since Haskell does not do it automatically.

Definition 7 (**Isomorphism**) An *isomorphism* between two data types A and B consists of two functions $f :: A \to B$ and $g :: B \to A$ with the following properties:

$f \circ g = id_B \wedge g \circ f = id_A$

□

Isomorphisms are very useful in the context of generic programming, because they allow us to convert between a small family of datatypes and another, much larger, family of datatypes. Implicit coercions provided in Scala, can be used to define isomorphisms that are automatically applied by the compiler. Alternatively we can encode an isomorphism as:

```
trait Iso[a, b] {
  def from (x : a) : b
  def to (x : b) : a
}
```

(Note, however, that the properties required by the isomorphism need to be verified manually since Scala does not provide any facilities for automatic verification.) We shall use this alternative definition in what follows since it allows us to relate more clearly our and Hinze's work, and it gives us an extra degree of flexibility in terms of which isomorphism to use.

Because GM is a based on type classes, it is not hard to translate Hinze's work into Scala — as we have seen in Section 2.1.6, there is a close connection between Scala's parametrized traits/classes and Haskell's type classes (Odersky, 2006a,b). The only apparent difficulty is that Hinze's approach relies on constructor classes (this is, type classes parametrized over a type constructor instead of a type). However, as we have already seen, this is also not a problem in Scala because we can encode type constructors using abstract types.

We present the transliteration of the Church encoding version of Hinze's *Generic* class in Figure 4.3. The key idea is that instances of the trait *Generic* represent generic functions over sums of products. A generic function is defined by different cases for sums, products, the unit type and also a few built-in types such as *int* or *char*. For sums and products, which have type parameters, we need extra arguments that define the generic functions for values of those type parameters.

```
trait Generic {
  type G <: TypeConstructor
  def unit : G {type A = One}
  def int : G {type A = int}
  def char : G {type A = char}
  def plus[a, b] (a : G {type A = a}, b : G {type A = b}) : G {type A = Plus[a, b]}
  def prod[a, b] (a : G {type A = a}, b : G {type A = b}) : G {type A = Prod[a, b]}
  def view[a, b] (iso : Iso[b, a], a: ⇒ G {type A = a}) : G {type A = b}
}
```

Figure 4.3: The **trait** *Generic*.

The *view* case is used to adapt generic functions to existing datatypes that are not directly defined as sums of products. To define a *view* case, we are required to provide an isomorphism between the datatype b and its corresponding sum of products a. We are also required to provide a value a: $\Rightarrow$ G{**type** $A = a$}, which is just an instance of the generic function for the isomorphic sum of products. Note that "$\Rightarrow$" before the parameter a signals that values of that type are *lazy*. This is needed because recursive types unfold infinitely.

In order to use generic functions defined with instances of *Generic*, we need to combine different cases that match the isomorphic sum of product of our datatype. For example, suppose that we wanted to use a generic function on a pair consisting on an integer and a character. To get the generic function at that type we would need:

```
def repP[g <: TypeConstructor] (implicit gen : Generic {type G = g}) =
  gen.prod (gen.int, gen.char)
```

It can be quite tedious to define these so-called *type representations*; however, since they are merely reflecting the structure of types, the compiler can automatically generate this code. In Figure 4.4 we show how this can be achieved in Scala. Again, this is almost a transliteration of Haskell's type class version, except that instead of using type classes we now make use of Scala implicit parameters to achieve the same effect; and we also need to mark the type parameter T with a contravariance annotation. The trait *Rep* defines a method *rep*, which takes an instance of *Generic* (a generic function) and returns a representation that can be used on a type T. We define representations for basic types and sums of products by using the corresponding methods in *Generic*. Parametrized representations, such as *RPlus* and *RProd* have one argument for each parameter that is itself a representation and can be implicitly passed.

```
trait Rep[−T] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) : g {type A = T}
}
implicit def RUnit = new Rep[One] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) = gen.unit
}
implicit def RInt = new Rep[int] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) = gen.int
}
implicit def RChar = new Rep[char] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) = gen.char
}
implicit def RPlus[a, b] (implicit a : Rep[a], b : Rep[b]) = new Rep[Plus[a, b]] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) =
    gen.plus (a.rep (gen), b.rep (gen))
}
implicit def RProd[a, b] (implicit a : Rep[a], b : Rep[b]) = new Rep[Prod[a, b]] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) =
    gen.prod (a.rep (gen), b.rep (gen))
}
```

Figure 4.4: Representations for generic functions.

4.3.2 Representations of Visitors

We have seen how to set up the generic machinery in Scala. We now show how to use our visitors/datatypes with the generic library.

Consider using generic functions over the parametric lists defined in Section 3.8.1. We need to create a representation of the list datatype as a sum of products. We can create the representation using the *view* method of *Generic*. However, we first need to define the isomorphism between parametric lists and sums of products. In Figure 4.5 we show how to define this isomorphism. We define an auxiliary type synonym *ListF* to represent the isomorphic sum of product type and implement the two methods *from* and *to* from the trait *Iso*. The *from* method is defined by case analysis using an external list visitor. The *to* method uses the visitor for sums to convert between the sum of products and lists.

Having defined the isomorphism, we can define a method *listRep* that constructs a representation for lists. For parametric types, the representation function follows the arity of the parametric datatype: if the datatype has *n* type parameters, the function giving the representation for that

```
def listIso[a] = new Iso[List[a], Plus[One, Prod[a, List[a]]]] {
  type ListF = Plus[One, Prod[a, List[a]]]
  def from (l : List[a]) = l.accept[External, ListF] (new VList[External, a, ListF] {
    def nil : ListF                  = Inl (One)
    def cons (x : a, xs : Rec) : ListF = Inr (Prod (x, xs.get))
  })
  def to (x : ListF) = x.accept (new PlusVisitor[One, Prod[a, List[a]], List[a]] {
    def inl (x : One)                = Nil[a]
    def inr (x : Prod[a, List[a]]) = Cons (x.fst, x.snd)
  })
}
```

Figure 4.5: Isomorphism between parametric lists and sums of products.

datatype will have *n* representation arguments. Therefore, for lists, we need to provide the representation for the type parameter as an argument to *listRep*. The method creates the representation by using the *view* case in *Generic* and providing it with *listIso* and the representation for the sum of products.

```
def listRep[a, g <: TypeConstructor]
  (a : g {type A = a}) (implicit gen : Generic {type G = g}) : g {type A = List[a]} =
  gen.view (listIso, gen.plus (gen.unit, gen.prod (a, listRep[a, g] (a) (gen))))
```

In order to integrate this with our library we should also provide an instance for *Rep*[*List*[*a*]], which will implicitly pick a representation for lists provided that an implicit representation for the type of the list parameter exists.

```
implicit def RList[a] (implicit a : Rep[a]) = new Rep[List[a]] {
  def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) =
    listRep[a, g] (a.rep (gen)) (gen)
}
```

A similar amount of work needs to be repeated for a new datatype. However, this code is mostly boilerplate and, with a smart enough compiler, it is possible to automatically generate it. Mechanisms that generate code for data types for use with generic programming include *Derivable Type Classes* (Hinze and Peyton Jones, 2000), *Scrap your Boilerplate* (Lämmel and Peyton Jones, 2003, 2004) or *Generic Clean* (Alimarine and Plasmeijer, 2001).

```
sealed case class List[a]
case class Nil[a] extends List[a]
case class Cons[a] (x : a, xs : List[a]) extends List[a]
def listIso[a] = new Iso[List[a], Plus[One, Prod[a, List[a]]]] {
  type ListF = Plus[One, Prod[a, List[a]]]
  def from (l : List[a]) : ListF = l match {
    case Nil ()       ⇒ Inl (One)
    case Cons (x, xs) ⇒ Inr (Prod (x, xs))
  }
  def to (x : ListF) = x.accept (new PlusVisitor[One, Prod[a, List[a]], List[a]] {
    def inl (x : One)                = Nil[a]
    def inr (x : Prod[a, List[a]]) = Cons (x.fst, x.snd)
  })
}
```

Figure 4.6: Isomorphism between a *List* case class and sums of products.

4.3.3 Representations of Scala's Case Classes

In the previous section we have shown how we can use our generic library on visitors by providing representations for those visitors. However, our generic programming library has a wider applicability and can be used with other kinds of hierarchies such as, for example, Scala's case classes. We will show again how we could represent lists, but this time around we will use case classes instead of a visitor for the source datatype.

In Figure 4.6 we define a form of parametric lists using case classes. The class *List*[*a*] represents the datatype and the classes *Nil* and *Cons* play the roles of the data constructors. The method *listIso*, like the equivalent method in Section 4.3.2, establishes and isomorphism between *List*[*a*] and the corresponding sum-of-product equivalent. This method is not very different from the one used for visitors. The only difference is, basically, that instead of using the *accept* method as a means to perform case analysis in *from*, with case classes we can use Scala's existing case analysis mechanism directly.

The equivalent *listRep* and *RList* methods for our *List* case class have, essentially, equal definitions to the visitor corresponding methods:

```
def listRep[a, g <: TypeConstructor] (a : g {type A = a})
                                     (implicit gen : Generic {type G = g}) =
  gen.view (listIso, gen.plus (gen.unit, gen.prod (a, listRep[a, g] (a) (gen))))
implicit def RList[a] (implicit a : Rep[a]) = new Rep[List[a]] {
```

```
    def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) =
      listRep[a, g] (a.rep[g] (gen)) (gen)
}
```

This demonstrates that the generic programming library can be used with the visitor library, but it is independent from it. In other words, there is nothing stoping us from using the generic programming library with standard Scala's class hierarchies like the ones defined by case classes.

4.3.4 Defining Generic Functions

Suppose we want to count the number of values contained in some structure. To do so we need create an instance of *Generic* and provide an abstract type *G* that essentially defines the type signature of the generic function in question. For our example we use *G = Size*:

```
trait Size extends TypeConstructor {
  def size (x : A) : int
}
```

The trait *Size* defines a method *size* that takes a value with the type of the structure we want to consume (*A* is the type parameter of the type constructor) and returns an integer.

We present the definition for the generic function (a subtype of *Generic*) in Figure 4.7. For each case, we define a new instance of *Size* specifying the behaviour of the function for that case. Although the function is supposed to count values of certain types, we want to make it generic enough such that we can control which values we want to count. Because of this option, we count 0 for basic values such as *unit*, *int* or *char* (this can later be overridden). For containers with sums and products we do the obvious thing, just calling *size* on each of the injections of the sum, and adding up the results of counting the two components of the product. Finally, the *view* case uses the isomorphism to provide the generic functionality over user-defined data types.

We define *MySize* as a trait instead of an object so that we can, in the future, extend a it. We shall look into this with more detail in Sections 4.3.5 and 4.3.6 and see that this yields increased reusability benefits. We may, however, be interested in having an object that simply inherits the functionality defined in *MySize*. Furthermore, this object can be made implicit so that methods like *rep* can automatically be fed with this instance of *Generic*. The object *mySize* does this:

```
implicit object mySize extends MySize
```

```
trait MySize extends Generic {
  type G = Size
  def unit = new Size {type A = One; def size (x : A) = 0}
  def int = new Size {type A = int; def size (x : A) = 0}
  def char = new Size {type A = char; def size (x : A) = 0}
  def plus[a, b] (a : G {type A = a}, b : G {type A = b}) = new Size {
    type A = Plus[a, b]
    def size (x : A) = x.accept (new PlusVisitor[a, b, int] {
      def inl (y : a) = a.size (y)
      def inr (z : b) = b.size (z)
    })
  }
  def prod[a, b] (a : G {type A = a}, b : G {type A = b}) = new Size {
    type A = Prod[a, b]
    def size (x : A) = x.accept (new ProdVisitor[a, b, int] {
      def prod (y : a, z : b) = a.size (y) + b.size (z)
    })
  }
  def view[a, b] (iso : Iso[b, a], a: ⇒ G {type A = a}) = new Size {
    type A = b
    def size (x : A) = a.size (iso.from (x))
  }
}
```

Figure 4.7: Defining a generic function for counting values.

We can then create a method *gsize* that provides an easy-to-use interface for the generic function: *gsize* takes some value of a type *t* (where *t* is representable) and returns the number of elements counted. However, as mentioned before, just using *mySize* as a parameter for *rep* will not give us a very useful generic function because it will always return 0 (since every base-type value will be counted as 0).

```
def gsize[t] (x : t) (implicit r : Rep[t]) : int =
  r.rep (mySize).size (x)
```

4.3.5 Reuse via Inheritance

The trait *MySize* defines a template for functions that count values of a certain type; however if no functionality is overridden (like in the object *mySize*), the resulting generic function does not count anything. The trait *MySize* becomes more useful when some of its functionality is overridden. In object-oriented languages the inheritance mechanism for defining new generic functions, by

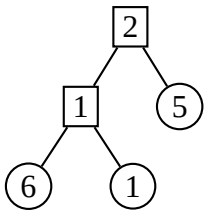


Figure 4.8: Tree with depth information.

overriding functionality of other generic functions. In languages without inheritance (like Haskell) this kind of reuse is more difficult to achieve.

Suppose that we wanted to define a generic function that counts all the integers in some structure. Using inheritance, all we have to is to extend *MySize* and override the case for integers so that it counts 1 for each integer value it finds.

```
object mySize2 extends MySize {
  override def int = new Size {type A = int; def size (x : A) = 1}
}
def countInt[t] (x : t) (implicit r : Rep[t]) : int =
  r.rep (mySize2).size (x)
```

With *mySize2* we can define a method *countInt*, which counts all the integers for some representable structure of type *t*.

Using generic functions is straightforward. The following snippet of code defines a list of integers *test* and a function *countTest* which applies *countInt* to this list.

```
def test = Cons (3, Cons (4, Cons (5, Nil[int])))
def countTest = countInt (test)
```

Note that the **implicit** parameter for the type representations is not needed, because it can be inferred by the compiler (since we provided an **implicit object** *RList*).

4.3.6 Local Redefinition

Suppose that we have an instance of some parametric datatype (like lists) and that we want to count how many values of the parametric type are in the some structure of that datatype. It is not possible to provide an implementation of *Generic* that defines such a function directly, because *Generic* cannot distinguish values of parametric types from other values that are just stored in the structure. For example, we could have a parametric binary tree that has an auxiliary integer at

each node that is used to store the depth of the tree at that node (this could be useful to keep the tree balanced). In Figure 4.8 we show one example of such binary trees: the squares represent the auxiliary integers and the circles represent the values that are contained in the tree. If the elements of the tree are integers and we try to use our *countInt* method we would count all the elements plus all the auxiliary integers, which may be unintended.

```
def testTree = Fork(2, Fork(1, Value(6), Value(1)), Value(5))
def five = countInt(testTree)   // returns 5
```

To solve this problem we remind ourselves that for parametric types we need to account for the representations of the type parameters. The method *listRep*, for example, needs to receive as an argument a representation of type $g\ \{\textbf{type}\ A = a\}$ for its type parameter. A similar thing happens with our binary trees. Assuming that the equivalent method is called *btreeRep*, we can provide a special-purpose counter for our trees that counts only the values of the type parameter.

```
def countA[a] = new Size {   // counts one for each element
  type A = a
  def size(x : a) = 1
}
def countBTree[a](x : BTree[a]) = btreeRep[a, Size](countA[a]).size(x)
def three = countBTree(test)   // returns 3
```

The basic idea here is that we replace the default (i.e. implicitly provided by the compiler) functionality that we would use for the type parameter by a user-defined one given by *countA* (which just counts one per value it sees).

4.4 GM and Indexed VISITORS

In this section we observe that the GM implementation is an instance of the VISITOR pattern: the trait *Generic* is the visitor and the trait *Rep* is the composite. However, unlike other visitors that we have seen so far, GM cannot be defined in terms of our visitor library. The reason for that is that the *GM* implementation is based on an *indexed visitor*: the recursive occurrences of the datatype have different types. We show how we could generalize our library to support a form of indexed visitors (that supports GM) and, by doing that, we also show how to remove one design choice from the original GM approach.

4.4.1 Indexed Visitors

Our visitor library is capable of expressing a very large family of visitors (or datatypes). For example we can use it to define mutually recursive types or many parametric container types (like lists or trees). In essence the kind of parametric types that can be defined have the property that all the self references in the definition of those types are equal. However, certain classes of parametric datatypes do not have that property. An example of such a class of datatypes are the so-called *nested datatypes*. The following datatype (written in Haskell's syntax) shows one example of a nested datatype:

data *PTree a* **where**
  PNil :: *PTree a*
  PFork :: $a \rightarrow$ *PTree* $(a, a) \rightarrow$ *PTree a*

The *PFork* case of a *PTree a* depends on a different instance *PTree* (a, a), which makes *PTree* not expressible in our library.

With nested datatypes, all the result references (the last type on the signature of the constructor) are the same as the datatype being defined; only references in argument positions can be different. A more general class of datatypes is the so-called *generalised algebraic datatypes* (GADTs) where that restriction is dropped, and it is also possible to have existential type variables — we have already seen, in Section 3.8.3, that our visitor library supports existential datatypes. The following datatype is an example of a GADT:

data *Rep t* **where**
  RUnit :: *Rep One*
  RInt :: *Rep Int*
  RProd :: *Rep a* $\rightarrow$ *Rep b* $\rightarrow$ *Rep* (*Prod a b*)
  RPlus :: *Rep a* $\rightarrow$ *Rep b* $\rightarrow$ *Rep* (*Plus a b*)

Note the similarity between *Rep t* and the traits *Generic* and *Rep* that we have defined in Figures 4.3 and 4.4. This is not coincidental: the original GM implementation was inspired by encodings of a datatype like this. Essentially the trait *Generic* is the visitor and the trait *Rep* is the composite. This datatype is not encodable using our visitor library, because we have references to *Rep* like *Rep One* or *Rep* (*Prod a b*).

```
trait TypeConstructor {type A}
  trait Strategy {
    type V <: Visitor
    type X <: TypeConstructor
    type T
    type Y
    def get : Y
  }
  trait Decompose[s <: Strategy] {
    def dec[v <: Visitor, x <: TypeConstructor, t]
      (vis : v {type X = x; type S = s}, comp : Composite[v, t]) :
        s {type V = v; type X = x; type T = t}
  }
  ...
  trait Visitor {
    type X <: TypeConstructor
    type S <: Strategy
    type R[v <: Visitor, t] = S {type X = Visitor.this.X; type T = t; type V = v}
  }
  trait Composite[-v <: Visitor, t] {
    def accept[s <: Strategy, x <: TypeConstructor] (vis : v {type X = x; type S = s})
      (implicit decompose : Decompose[s]) : x {type A = t}
  }
  trait Internal extends Strategy {type Y = X {type A = T}}
  trait External extends Strategy {type Y = Composite[V, T]}
```

Figure 4.9: A Visitor library with support for unnested GADTs

4.4.2 A Visitor Library for Indexed Visitors

We now show how to modify our library so that it supports a larger family of datatypes, namely a form of indexed datatypes. We shall call this family of datatypes the (one parameter) *unnested GADTs*. With this new library it will be possible to easily define *Rep* but *PTree* is problematic because of the complex recursion pattern involved (see Bird and Meertens (1998); Bird and Paterson (1999); Martin *et al.* (2004) for more details). In other words, this family allows us to refine the return types of the constructors and to have existential variables in the contructors, but, while we can define recursive references with refined type arguments (in the style of nested datatypes), it is not trivial (or it may be impossible) to define the constructors for nested datatypes.

The functional specification of the new version of our visitor library follows from the following composite:

$$Composite\ V\ T \equiv \overbrace{\forall X\ S.\ Decompose\ S \Rightarrow \underbrace{V\ (S\ V\ X)\ X}_{Visitor} \Rightarrow X\ T}^{accept\ method}$$

Compared to the *Composite* presented in Section 3.6 the differences are that the new *Composite* is now parametrized by a type T and that the return type of the *accept* method is refined to $X\ T$. Also X is now a type constructor rather than just a type. In Appendix F the full Haskell code corresponding to the functional specification is presented.

Figure 4.9 shows a modified version of our visitor library supporting unnested GADTs. Basically, the modifications consist of adding an extra type parameter (that appears in the code as t or T) in a few places, and refining some types. This extra type parameter is the (type) index of the datatype. More specifically, we can see that *Strategy* has now an extra abstract type T and X is refined into a *TypeConstructor*. The definition of *dec* in *Decompose* is modified to include an extra type parameter t and slightly refine its signature. In the *Visitor*, the type X is refined to a *TypeConstructor*, and the type R (representing the type of recursive references) is now parametrized by t. The *Composite* has an extra type argument t, and the *accept* method refines its return type. Finally, *Internal* and *External* refines Y with T.

4.4.3 GM as an Instance of the Visitor Library

It is now time to show how the essence of GM is itself an instance of the visitor pattern and how we can use the indexed version of the visitor library to capture it. As we shall see, *Generic* is just a visitor component and *Rep* is the corresponding composite. To show that *Generic* is a visitor component we need to find a suitable $V = Generic$ that can be replaced in the *Composite* equation in Section 4.4.2 so that $Rep\ T$ becomes an instance of $Composite\ Generic\ T$. Such an instantiation of V is given next:

$$\begin{aligned} Generic\ S\ X &\equiv X\ One \times X\ Int \times (\forall A\ B.\ S\ A \Rightarrow S\ B \Rightarrow X\ (Pair\ A\ B)) \times \ldots \\ Rep\ T \quad &\equiv Composite\ Generic\ T \\ &\equiv \forall S\ X.\ Decompose\ S \Rightarrow Generic\ (S\ Generic\ X)\ X \Rightarrow X\ T \end{aligned}$$

Generic consists of a product of functions satisfying the visitor requirements. Each element of the product represents a case of the generic function as explained before. For parametrized cases (that

is, for cases involving type constructors), we have, like with the definition of *Generic* in Figure 4.3, one argument for each type parameter of the type constructor. However, the thing to note is that the type of those arguments are more general than the ones in Figure 4.3. It is this extra generality that allows us to parametrize *Generic* by the decomposition strategy allowing us to obtain, for example, the two implementations presented by Hinze (2004).

For (non-parametrized) types we can provide representations for some type T by giving the corresponding instance of a function $rep_T \in Rep\ T$. For example, for *One* and *Int* we would have:

$$
\begin{aligned}
&rep_{One} \in Rep\ One \\
&rep_{One}\ (unit \times int \times prod \times \ldots) \equiv unit \\
&rep_{Int} \in Rep\ Int \\
&rep_{Int}\ (unit \times int \times prod \times \ldots) \equiv int
\end{aligned}
$$

For parametrized types we need to provide representations for each of the constructors. If we have a binary type constructor T, then the corresponding representation function has type $rep_{TAB} \in Rep\ A \Rightarrow Rep\ B \Rightarrow Rep\ (T\ A\ B)$. We exemplify with a representation for pairs:

$$
\begin{aligned}
&rep_{PairAB} \in Rep\ A \Rightarrow Rep\ B \Rightarrow Rep\ (Pair\ A\ B) \\
&rep_{PairAB}\ ra\ rb\ (unit \times int \times prod \times \ldots) \equiv prod\ (dec_S\ (unit \times int \times prod \times \ldots)\ ra) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (dec_S\ (unit \times int \times prod \times \ldots)\ rb)
\end{aligned}
$$

More generaly, given some type constructor T with arguments $A_1\ \ldots\ A_n$ then, we can create a representation for $T\ A_1\ \ldots\ A_n$ as follows:

$$
\begin{aligned}
&Generic\ S\ X \equiv \ldots \times (\forall A_1\ \ldots\ A_n.\ S\ A_1 \Rightarrow \ldots \Rightarrow S\ A_n \Rightarrow X\ (T\ A_1\ \ldots\ A_n)) \times \ldots \\
&rep_{TA_1 \ldots A_n} \in Rep\ A_1 \Rightarrow \ldots \Rightarrow Rep\ A_n \Rightarrow Rep\ (T\ A_1\ \ldots\ A_n) \\
&rep_{TA_1 \ldots A_n}\ ra_1\ \ldots\ ra_n\ (\ldots \times t \times \ldots) \equiv t\ (dec_S\ (\ldots \times t \times \ldots)\ ra_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (dec_S\ (\ldots \times t \times \ldots)\ ra_n)
\end{aligned}
$$

The *rep* function is, in Hinze (2004) words, "*the mother of all generic functions*" and, as Hinze (2000) himself shows, "*generic functions possess polykinded types*". Our general account of generic representations, although not as general as the scheme shown by Hinze (because we do not consider type parameters of higher kinds), follows the same basic principles.

```
trait Generic extends Visitor {
  type Rec[t] = R[Generic, t]
  def unit  : X {type A = One}
  def int   : X {type A = int}
  def char : X {type A = char}
  def plus[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Plus[a, b]}
  def prod[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Prod[a, b]}
  def view[a, b] (iso : Iso[b, a], a: ⇒ Rec[a]) : X {type A = b}
}
type Rep[T] = Composite[Generic, T]
```

Figure 4.10: GM as a Visitor

In Scala Figure 4.10 shows how to define *Generic* and *Rep* using the modified library. The new version of *Generic* is very similar to the one presented in Figure 4.3, except that *G* is now called *X* and the arguments of the constructors use *R* instead of *G* (or *X*). Regarding the composite *Rep*, the difference is that the method *rep* is now called *accept*.

Defining generic functions proceeds almost in the same way as before. Figure 4.11 shows a partial definition of *MySize*. Compared to the version presented in Figure 4.3, we need to set the extra abstract type *S* (representing the strategy) to *Internal*, and use the method *get* to extract the values from the strategy. Both modifications would be unnecessary had we also reimplemented the notational enhancements that we presented in Section 3.7.

The GM version defined using the modified visitor library strictly generalizes the one presented in Section 4.3, because we can not only have internal visitors, but also use other kinds of visitors (such as external or paramorphic visitors). Like with other datatypes, this fact can be a valuable advantage because it does not force us to the design choice of the kind of visitor upfront; instead, for each generic function that we define, we can decide which strategy (or kind of visitor) we want to use. In contrast, in Hinze (2004) two alternative implementations of *Generic* are presented (one based on a Church encoding and another based on a Parigot encoding) forcing us into a design choice.

```
trait MySize extends Generic {
  type X = Size
  type S = Internal
  def int = new Size {type A = int; def size (x : A) = 0}
  ...
  def prod[a, b] (a : R[a], b : R[b]) = new Size {
    type A = Prod[a, b]
    def size (x : A) = x.accept (new ProdVisitor[a, b, int] {
      def prod (y : a, z : b) = a.get.size (y) + b.get.size (z)
    })
  }
  def view[a, b] (iso : Iso[b, a], a: ⇒ R[a]) = new Size {
    type A = b
    def size (x : A) = a.get.size (iso.from (x))
  }
}
```

Figure 4.11: Generic function using the *Generic* visitor.

4.5 A Visitor for a Family Based on Sums of Products

Our presentation (and the classical one) of the Visitor pattern uses products of functions, where each function corresponds to a case of a datatype. This nicely matches the familiar notion of pattern matching in functional languages, which allows us to define functions intuitively. However, writing generic functions for products of functions is difficult. The generic programming techniques that we used in the previous sections allow us to write generic functions on sums of products, which force us to do expensive conversions between products of functions and sums of products like the ones in Figure 4.5. In this section we will propose a different family of visitors that can be defined as a concrete visitor using our existing library giving us an alternative *view* on datatypes. With this new family we can define visitors in terms of sums of products directly.

4.5.1 A Visitor Based on Sums of Products

Writing generic functions for products of functions is difficult. A solution is to first map our visitors into sums of products and then apply well known DGP techniques that work for that family of datatypes. This was the approach we have taken in the previous sections. However, there are

inconveniences in doing it this way: firstly it hinders performance (because we need to map between the visitor and the sum of products); secondly, without compiler support, it involves some boilerplate code for each visitor; finally, because the recursion pattern is not explicit, it is harder to reason about those visitors.

A Functional Specification

We have already observed (in Section 3.5.1) that a family of datatypes based on sums of products can be expressed using products of functions (since $(F\ R \Rightarrow X) \Rightarrow X$ is just a particular case of $V\ R\ X \Rightarrow X$, where $V\ R\ X \equiv F\ R \Rightarrow X$). Defining

$$\begin{aligned} &\mathit{SumProdF}\ F\ R\ X \equiv F\ R \Rightarrow X \\ &\mathit{SumProd}\ F \qquad\quad \equiv \mathit{Composite}\ (\mathit{SumProdF}\ F) \end{aligned}$$

we can capture the family of visitors based on sums of products as a concrete visitor of our library. There are single (generic) constructor and deconstructor functions for this visitor given by

$$\begin{aligned} &\mathit{inF} \quad \in \mathit{Functor}\ F \Rightarrow F\ (\mathit{SumProd}\ F) \Rightarrow \mathit{SumProd}\ F \\ &\mathit{inF}\ t \equiv \lambda v \Rightarrow v\ (\mathit{fmap}\ (\mathit{dec}\ v)\ t) \\ &\mathit{outF} \in \mathit{Functor}\ F \Rightarrow \mathit{SumProd}\ F \Rightarrow F\ (\mathit{SumProd}\ F) \\ &\mathit{outF} \equiv \mathit{dec}\ (\mathit{fmap}\ \mathit{inF}) \end{aligned}$$

which form an isomorphism (that is $\mathit{inF} \circ \mathit{outF} \equiv \mathit{id}$ and $\mathit{outF} \circ \mathit{inF} \equiv \mathit{id}$). Here we assume that F is a functor and that $\mathit{Functor}\ F$ is implicitly passed (see Section 2.2.1 for a definition of *Functor*). We refer to Gibbons (2006) for more details about this family of visitors. The code for the functional specification in Haskell is presented in Annexe G.

Scala Implementation

Figure 4.12 shows a visitor for datatypes based on sums of products and implemented with our generic library. The visitor component (the trait *SumProdVisitor*) consists of a single method that takes as a first parameter a functor F (based on sums of products) and, as the second parameter an (implicit) *Functor* object that describes how to perform the functorial map over F. The composite component *SumProd* is just a type synonym to $\mathit{Composite}[\mathit{SumProdVisitor}[F]]$. Finally (and like all other visitors) *VSumProd* can be used as a shorthand to define visitors on sums of products with a functional notation.

The single constructor *in* (that follows from the single *visit* method) can be defined generically

```
trait SumProdVisitor[F <: TypeConstructor] extends Visitor {
  type Rec = R[SumProdVisitor[F]]
  def visit (x : F {type A = Rec}) : X
}
type SumProd[f <: TypeConstructor] = Composite[SumProdVisitor[f]]
abstract class VSumProd[s <: Strategy, f <: TypeConstructor, b]
  (implicit decompose : Decompose[s])
  extends VisitorFunc[SumProdVisitor[f], s, b] (decompose)
  with SumProdVisitor[f]
```

Figure 4.12: A visitor for sums of products data types.

(on the sum of products functor f) using the functorial map that traverses the structure and applies *dec* to each recursive occurrence.

```
def in[f <: TypeConstructor] (x : f {type A = SumProd[f]})
    (implicit funct : Functor[f]) : SumProd[f] = new SumProd[f] {
  def accept[s <: Strategy, x] (vis : SumProdVisitor[f] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.visit (funct.fmap ((y : SumProd[f]) ⇒ decompose.dec (vis, y)) (x))
}
```

The deconstructor *out*, the inverse of *in*, can also be encoded in Scala as follows:

```
def out[f <: TypeConstructor] (s : SumProd[f]) (implicit funct : Functor[f]) =
  s.accept (new VSumProd[Internal, f, f {type A = SumProd[f]}] {
    def visit (x : f {type A = R[SumProdVisitor[f]]}) =
      funct.fmap ((y : R[SumProdVisitor[f]]) ⇒ in (y.get) (funct)) (x)
})
```

4.5.2 Creating New Datatypes

In order to create a new data type with a sum of products visitor, we need to define the type constructor based on sums of products that represents the datatype and the corresponding functor instance. In Figure 4.13 we see a definition of lists. The trait *ListF* — or more precisely the method *listF* in that trait — is used in Scala to define the sum of products based type constructor representing the datatype. Having this, the new type for parametric lists can be defined in terms of *SumProd*[*ListF*[*a*]]. Because our type constructor is based on sums of products it defines a functor — the method *listFunctor* provides the instance of *Functor* for *ListF*. Finally, the list constructors *Nil* and *Cons* can be defined in terms of *in*.

```
trait ListF[a] extends TypeConstructor {
  def listF : Plus[One, Prod[a, A]]
}
type List[a] = SumProd[ListF[a]]
implicit def listFunctor[p] = new Functor[ListF[p]] {
  def fmap[a, b] (f : a ⇒ b) (x : ListF[p] {type A = a}) : ListF[p] {type A = b} = new ListF[p] {
    type A = b
    def listF = x.listF.accept (new PlusVisitor[One, Prod[p, a], Plus[One, Prod[p, b]]] {
      def inl (y : One)          = Inl[One, Prod[p, b]] (y)
      def inr (z : Prod[p, a]) = Inr[One, Prod[p, b]] (Prod (z.fst, f (z.snd)))
    })
  }
}
def Nil[a] : List[a] = in[ListF[a]] (new ListF[a] {
    type A = List[a];
    def listF = Inl[One, Prod[a, A]] (One)
  })
def Cons[a] (x : a, xs : List[a]) : List[a] = in[ListF[a]] (new ListF[a] {
    type A = List[a];
    def listF = Inr[One, Prod[a, A]] (Prod (x, xs))
  })
```

Figure 4.13: Parametric lists using a sum of products visitor.

Comparing the sums of products with the product of functions version of lists in Section 3.8.1 we can see that we have to provide roughly the same amount of code. The differences consist basically in the way that we deal with constructors in each approach. With products of functions, each constructor defines the traversal code itself. With sums of products, we use the functor (which we have to define) to perform the traversal.

While no big differences between the two versions exist for setting up the code for a new datatype, some more substantial differences exists in the code necessary to enable generic programming. Compare the following

```
def listFIso[p, a] = new Iso[ListF[p] {type A = a}, Plus[One, Prod[p, a]]] {
  def from (x : ListF[p] {type A = a}) : Plus[One, Prod[p, a]] = x.listF
  def to (x : Plus[One, Prod[p, a]]) = new ListF[p] {type A = a; def listF = x}
}
```

with the code that we had to provide in Figure 4.5. Because we use sums of products directly to define our lists the isomorphism is straightforward to define, which can be seen as a small advantage

```
def RSumProd[f <: TypeConstructor] (implicit f : FRep[f], funct : Functor[f]) =
  new Rep[SumProd[f]] {
    def rep[g <: TypeConstructor] (implicit gen : Generic {type G = g}) =
      gen.view (sumProdIso[f], f.frep[g, SumProd[f]] (rep[g] (gen)) (gen))
  }
def sumProdIso[f <: TypeConstructor] (implicit funct : Functor[f]) =
  new Iso[SumProd[f], f {type A = SumProd[f]}] {
    def to (x : f {type A = SumProd[f]}) = in (x)
    def from (x : SumProd[f]) = out[f] (x)
  }
```

Figure 4.14: A representation for sums of products visitors in Scala.

of this version. More importantly, since sums of products are used directly, we do not incur on any major performance penalties derived from the embedding-projection pairs (*from* and *to*).

4.5.3 Functorial Representations

Because sum of product visitors are based on functors, we need to have functor representations (which differ from normal type representations). The trait *FRep* that serves that purpose is defined as:

```
trait FRep[f <: TypeConstructor] {
  def frep[g <: TypeConstructor, a]
    (a : g {type A = a}) (implicit gen : Generic {type G = g}) : g {type A = f {type A = a}}
}
```

The method *frep* takes a representation of the type argument of the functor and an (implicit) instance of *Generic* and returns the representation for the functor. Note that *frep* can be seen as a generalization of the method *listRep* that we presented in Section 4.3.2.

Given that we can represent functors, we can now define representations for our sum of products composites as shown in Figure 4.14. The *sumProdIso* method defines the isomorphism that converts between the *SumProd* composite and the actual sum of products element wrapped inside it. The embedding-projection pairs are respectively given by the *in* and *out* methods. Using this isomorphism, we can then create a *view* that represents any sum of products visitor that has a functorial representation.

Instances of *FRep* provide representations for the different functors. For example, with our lists

based on sums of products, we need to provide an instance of *FRep* for *ListF*[*p*].

```
def listFRep[p] (implicit p : Rep[p]) = new FRep[ListF[p]] {
  def frep[g <: TypeConstructor, a] (a : g {type A = a}) (implicit gen : Generic {type G = g}) =
    gen.view[Plus[One, Prod[p, a]], ListF[p] {type A = a}]
      (listFIso, gen.plus (gen.unit, gen.prod (p.rep[g] (gen), a)))
}
```

As we shall see in the next section, by viewing a recursive datatype as a sum of products functorial representation, we will be able to specify how a generic function behaves for the recursive occurrences of a datatype, by using local redefinitions.

4.5.4 Separating Recursion from Generic Programming

Our sum-of-products based visitors allow us to define generic functions that separate the generic parts (the boilerplate associated with sums of products) from the recursion points. This separation gives us extra flexibility and makes reasoning about generic functions easier. As a simple example consider a function that counts the number of recursive occurrences in some value from some recursive datatype.

```
def mysize[f <: TypeConstructor] (x : SumProd[f]) (implicit fr : FRep[f]) =
  x.accept[Internal, int] (new VSumProd[Internal, f, int] {
    def sizea = new Size () {
      type A = Rec
      def size (x : Rec) : int = x + 1
    }
    def visit (x : f {type A = Rec}) = fr.frep[Size, Rec] (sizea).size (x)
  })
```

The method *mysize*, given a value *x* (which is an element of a sum-of-products datatype), is defined using a sum-of-products visitor on *x*. The method *sizea* is an example of a local redefinition (presented in Section 4.3.6): this method becomes the argument of *frep* that specifies what the function should do for recursive occurrences. Since the goal of this function is to count the number of recursive occurrences, the method *sizea* is simply the successor function. We give a brief example of the usage of *mysize* next:

```
val test = Cons (1, Cons (2, Cons (3, Nil[int])))
def sizeList[a] (x : List[a]) (implicit a : Rep[a]) = mysize (x) (listFRep)
```

The value *test* defines a three element list. The method *sizeList* takes a list of values of type *a*

and, provided that there exists a representation for *a*, returns the number of recursive occurrences of that list. If we evaluate *sizeList* (*test*) we get 3 as the result.

Using *FRep* we can define a generic function that applies another generic function to the sum of product boilerplate, but redefines the behaviour of that function for recursive occurrences. This is not possible to do in general for visitors defined with our library; only the ones that are defined using the sum-of-products visitor allow this functionality. Because we have access to the recursion points, another thing we can do is capture well-known recursion patterns. We show how to capture catamorphisms and anamorphisms (Meijer *et al.*, 1991) next.

Catamorphisms (or *folds*) are possibly the most well-known recursion pattern, allowing us to write iterative definitions. The functional specification of catamorphisms is given by

$$cata \quad \in Functor\ F \Rightarrow (F\ A \Rightarrow A) \Rightarrow SumProd\ F \Rightarrow A$$
$$cata\ v\ m \equiv m\ v$$

and the corresponding Scala implementation is:

```
def cata[f <: TypeConstructor, b]
  (func : f {type A = b} ⇒ b) (comp : SumProd[f]) (implicit funct : Functor[f]) : b =
    comp.accept[Internal, b] (new VSumProd[Internal, f, b] {
      def visit (x : f {type A = Rec}) = func (funct.fmap[Rec, b] ((y : Rec) ⇒ y.get) (x))})
```

Note that internal visitors (for example, the visitor used by *mysize* above) are essentially catamorphisms: the body of the catamorphism (i.e. the parameter $func : f\ \{\textbf{type}\ A = b\} \Rightarrow b$) corresponds to the *visit* method. The only difference is that with catamorphisms, we do not need to apply the coercion between *Rec* and *b*, while using the internal visitor directly we do (even if this is done implicitly).

Anamorphisms (or *unfolds*) are the dual (in a categorical interpretation) recursion pattern of catamorphims and they capture a form of co-recursion. The functional specification of anamorphisms is given by

$$ana \in Functor\ F \Rightarrow (A \Rightarrow F\ A) \Rightarrow A \Rightarrow SumProd\ F$$
$$ana\ c\ x \equiv \lambda v \Rightarrow v\ (fmap\ (dec\ v\ .\ ana\ c)\ (c\ x))$$

and the corresponding Scala implementation is:

```
def ana[f <: TypeConstructor, b]
    (func : b ⇒ f {type A = b}) (x : b) (implicit funct : Functor[f]) : SumProd[f] =
      new SumProd[f] {
```

```
  def accept[s <: Strategy, x] (vis : SumProdVisitor[f] {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.visit (funct.fmap ((y : b) ⇒
      decompose.dec[SumProdVisitor[f], x] (vis, ana[f, b] (func) (y))) (func (x)))
}
```

4.6 Example: Generic Serialization and Deserialization

We are now in a position to develop a fully-fledged application using our library. In this section we will develop a simple serialization library — a common application of generic programming. The full code for this little library in presented in Annexe E.

For many applications we need to serialize information one way or another: it may be in some binary format, XML or some other textual format. Because different applications use different data structures, we usually need to define our own serialization/deserialization functions or hope that the language we use has some built-in support for that particular task. An alternative, which we will take in this section, is to write a generic function that supports a wide range of data structures.

In the next two subsections we will present, respectively, binary generic serializer and deserializer functions. These functions are proof of concept only, since the serialization process does not produce an actual binary stream but a string consisting of zeros and ones. Producing an actual binary stream should not, however, be too difficult.

4.6.1 Serialization

In the implementation of our generic serializer function (see Annexe E), the trait *Serialize*, as explained in Section 4.3.4, defines the type of the serialization function. The trait *MySerialize* (a subtype of *Generic*) defines the body of our generic function. Finally the object *mySerial* provides a default implementation.

The methods defined in *MySerialize* define the different cases of the generic function. A *unit* value does not need any bit to be represented, so we can return the empty String as a result of encoding such values. For primitive types like *int* and *char* we just assume that there exist functions *encodeInt* and *encodeChar* that serialize those values. For sum types, we require one bit to represent the choice between the two alternatives: the values based on *inl* are appended with zero; and the values based on *inr* are appended with one. With products the final result is given by the concatenation of the serializations of the first and second components. Finally the *view* case converts

the value into a sum of product and then invokes the serializer on the result.

We can provide a Scala method *serial* that provides an easy to use interface for our generic function. The method takes a representable value x (i.e. a value whose type t has a representation r of type $Rep[t]$) and calls the *rep* method with the default *mySerial* function, thus allowing us to call *serialize* on x.

```
def serial[t] (x : t) (implicit r : Rep[t]) : String = r.rep (mySerial).serialize (x)
```

Serializing Sum of Products Visitors

With *serial* all representable instances of *Visitor* can be serialized. Alternatively we could also have provided a serialization function that works just for sum-of-products visitors. It is interesting to look at this other solution because, unlike the solution above, here the recursion pattern (which is associated with the kind of visitor that is used) becomes explicit.

```
def serialSumProd[f <: TypeConstructor] (x : SumProd[f]) (implicit fr : FRep[f]) =
  x.accept[Internal, String] (new VSumProd[Internal, f, String] {
    def seriala = new Serialize () {
      type A = Rec
      def serialize (x : A) : String = x
    }
    def visit (x : f {type A = Rec}) = fr.frep[Serialize, Rec] (seriala).serialize (x)})
```

In this case we can see that the recursion pattern that is used is a catamorphism since we are using an internal visitor — we could make the recursion pattern even more explicit by using the *cata* method presented in Section 4.5instead. While with *serial* we do not need to specify what to do whenever a recursive occurrence appears, with this definition we do. The inner method *seriala* specifies that. Because we have an internal visitor, the recursive occurrences are replaced by the result of the recursive call and, therefore, what we need to do is just to return that result.

To give and example, consider the following list:

```
def testVal = Cons (3, Cons (4, Nil[int]))
```

Invoking the two serialization functions for this value is achieved with:

```
def testSerial = serial (testVal)
def testSumProd = serialSumProd (testVal)
```

Both invocations would give a result like:

```
10000000000000000000000000000011100000000000000000000000000001000
```

This depends, of course, on how the primitive serializers are implemented. In this case we opted (for readability), to encode integers using 16 bits.

4.6.2 Deserialization

In the implementation of our deserializer function (see Annexe E) the trait *DeSerialize* provides the type of our deserialization function; the trait *MyDeSerialize* defines the body of the generic function using *DeSerialize* as the return type of the generic functions; and the object *myDeSerial* provides a default implementation based on *MyDeSerialize*.

The *deserialize* method in *DeSerialize* decodes a segment of the input string and returns a piece of data plus the remainder of the string yet to be decoded. For the *unit* case, we know that there is nothing else to decode and thus we return the value *One* and an empty string. For primitive types we use standard deserialization functions. For the sum case of our generic function we need to test if the first character of the string is a zero or a one. If it is a zero we know we need to create an *Inl* value, otherwise we create an *Inr* value. For products, we start by decoding the segment of the string that corresponds to the first component of the product and then we decode the segment corresponding to the second component. We then return the product of the two results plus the remainder of the string. For the *view* case we deserialize the segment of the string corresponding to the isomorphic sum of products value; convert the resulting sum of product into the value of the data type and finally return that result plus the remainder of the string.

The method *deSerial* taking a string and an implicit parameter with a representation of some type *t* uses the default implementation of the generic function *myDeserial* to provide an easy-to-use interface to deserialization.

```
def deSerial[t] (x : String) (implicit r : Rep[t]) : t = r.rep (myDeSerial).deSerialize (x).fst
```

Deserialization for Sum of Products Visitors

We can also have a deserialization function for sum of products visitors. However, because this function produces a value of a data type instead of consuming data (like with serialization), we need to use a productive recursion pattern. We have introduced anamorphisms in Section 4.5, that can be used here:

```
def deserialAux = new DeSerialize {
  type A = String
  def deSerialize (x : String) : Prod[A, String] = Prod (x, "")
```

```
}
def deSerialSumProd[f <: TypeConstructor]
  (x : String) (implicit fr : FRep[f], funct : Functor[f]) =
    ana[f, String] ((y : String) ⇒
      fr.frep[DeSerialize, String] (deserialAux).deSerialize (y).fst) (x)
```

The auxiliary method *deserialAux* specifies what to do when we find recursive occurrences. The body of the method *deSerialSumProd* is an anamorphism, which consumes the string *x* and for each segment *y* (representing sums of products) of *x* applies the generic function *DeSerialize*.

4.7 Discussion

This chapter explains how we can use DGP techniques with our visitor library. The DGP technique that we use, inspired by GM, follows the current trend of lightweight approaches to generic programming, where genericity can be captured as a software component that can be used to define generic functions.

Generic functions are powerful software components on their own since they can be very flexibly parametrized, which allows them to be adapted to several problem domains (via the datatypes used in the solutions for those problems). Our example application is good example of generic functions that can be applied to a wide range of problem domains. Most software applications require some form of persistence in order to store the state of some piece data being manipulated by these same applications. A mechanism that serializes/deserializes data is almost a necessity since providing special purpose functions for this task is error-prone and tedious. For many programming languages the solution is to basically build in special support in the compiler for this task or to use some form of pre-processor that generates serialization and deserialization functions automatically. Alternatively, if the language supports some form of run-time introspection (for example, the reflection mechanism in Java), we can try to use that mechanism to define a form of generic functions that accomplishes this task. However, neither solution is completely satisfactory: built-in support tends to be too inflexible; and run-time introspection mechanisms are not type-safe and are penalizing in terms of performance. In contrast, the use of generic functions allows a lot of flexibility (specially when support for extensibility, presented in Chapter 5, is added) while being type-safe and having minor performance penalties.

In Haskell there has been a recent flurry of proposals for generic programming libraries (Cheney and Hinze, 2002; Hinze, 2004; Lämmel and Peyton Jones, 2005; Oliveira *et al.*, 2006; Hinze

et al., 2006; Weirich, 2006; Hinze and Löh, 2007), all of which having interesting aspects but none emerging as a clearly best option. Because of that, an international committee has been set up with the goal of developing a standard generic programming library in Haskell. The first effort from that committee was a thorough comparison of most of the generic programming libraries proposed (Jeuring *et al.*, 2007). The EMGM variation (Oliveira *et al.*, 2006) of GM, which is the basis of parts of this thesis, was in the pool of approaches compared. The Scala library proposed in this thesis has basically all the same advantages of that approach and a few extra ones. Firstly, with our approach we can, very easily, reuse one existing generic function to define a new one (this is demonstrated in Section 4.3.5). With the Haskell approaches, this kind of reuse is harder to achieve. The only mechanism that we know of that comes close, in terms of simplicity, to this form of reuse is Generic Haskell's *default cases* (Löh, 2004). Secondly, we show in Section 4.5, how we can use a different family of visitors within our generic programming framework. In essence this family of visitors is the so-called *fixpoint view* in Generic Haskell (Holdermans *et al.*, 2006). Therefore our approach has support for other views, which contrasts with the evaluation of EMGM which considered the approach as not having such support. However, this second advantage is not particular to the Scala implementation and it could be adapted to the Haskell one.

In Section 4.4 we present a modified version of our visitor library that can be used to define a class of type-indexed visitors. In particular, we show that the implementation of GM is itself a visitor and can be encoded with this modified library. In the original presentation of GM a design choice between Church and Parigot encodings — or, using the visitor terminology, a choice between internal and external visitors — has to be made. Since our visitors are naturally strategy generic this design choice is unnecessary. We believe that this provides a novel and interesting insight for the field of DGP since existing generic programming approaches always choose one particular encoding and necessarily inherit the disadvantages of that encoding, which would not be the case with our generic programming library. For example, in recent versions of GH, because a Church encoding is used, mutually-recursive generic functions are hard to define. The '*dependency style*' of GH Löh (2004) is an elegant, but complex, solution for the problem of these (mutual) dependencies and it does not solve other issues that stem from the Church encoding.

There are several other applications (apart from GM) for our indexed variation of the library. For instance, in Hinze (2003); Peyton Jones *et al.* (2006) many example applications for GADTs have been proposed, with most of them being examples of unnested GADTs. We could use our

library to define those examples and automatically benefit from recursion patterns (which can be considered themselves a form of generic functions). In previous work (Oliveira and Gibbons, 2005) we proposed a design pattern for type-indexed functions inspired by GM. There were three main variations of the design pattern, and two of the variations had equivalent expressive power. In essence we had the same variations as with the original GM or visitors in general. If we were to present that work today, we could have done it as a software component rather than a design pattern.

Chapter 5

Extensible Visitors and Generic Functions

In Chapter 4 we developed a generic programming library in Scala based on the GM approach and showed a possible implementation of this library as a visitor. With this library it is possible to write generic functions that work over a large family of visitors. However, the *generic functions* (i.e. the visitor components) have a single, non-extensible, definition. In contrast, *ad-hoc polymorphic* functions require separate implementations for each datatype, but can be extended with new cases at any time. The fact that generic functions cannot be extended is a severe drawback, because often we want to define some ad-hoc behaviour for new datatypes. This limitation precludes the design of an extensible and modular generic programming library. The problem of extensibility of generic functions is one particular instance of the so-called *expression problem*. In this chapter, we show that it is possible to develop extensible generic functions (and, more generally, extensible visitors) using our visitor library. We show two solutions with different trade-offs.

5.1 Introduction

We have seen, in Chapter 4, that a generic function is a function defined over the structure of types. With generic functions a single definition suffices to obtain a function that works for a large family of visitors. By contrast, an *ad-hoc polymorphic* function (Strachey, 1967) requires a separate implementation for each data type. In Haskell, for example, ad-hoc polymorphic functions are implemented using type classes. In Scala, the same effect can be achieved with parametric traits

```
trait Encode[t] {
  def encode (x : t) : String
}
implicit def EncodeChar = new Encode[Char] {
  def encode (x : Char) = encodeChar (x)
}
implicit def EncodeInt = new Encode[int] {
  def encode (x : int) = encodeInt (x)
}
implicit def EncodeList[a] (implicit encA : Encode[a]) = new Encode[List[a]] {
  def encode (x : List[a]) = x.accept (new VList[Internal, a, String] {
    def nil = "0"
    def cons (x : a, xs : Rec) = "1" + encA.encode (x) + xs
  })
}
```

Figure 5.1: An ad-hoc binary encoder

and implicit parameters.

In Figure 5.1 we show how to implement an ad-hoc polymorphic function for binary encoding. The trait *Encode* [*t*] defines a type-overloaded (on the type parameter *t*) function *encode*. The implicit definitions provide implementations of *Encode* [*t*] for a number of specific types *t*. The definition *EncodeList*[*a*] provides an implementation for *List*[*a*], and is only defined if there exists an implementation of *Encode* [*a*] — in other words, we can only encode lists if we know how to encode the elements. As with the generic serializer that we developed in Chapter 4, we assume that primitive bit encoders for integers and characters are given, respectively, by *encodeInt* and *encodeChar*. Lists are encoded by replacing an occurrence of the empty list *Nil* with the bit 0, and occurrences of the list constructor *Cons* with the bit 1 followed by the encoding of the head element and the encoding of the remaining list. The function *encode* thus works on characters, integers, and lists. If we call *encode*, the compiler figures out the correct implementation to use, or, if no suitable instance exists, it reports a type error, nicely mimicking the functionality of Haskell type classes.

The function *encode* can be extended at any time to work on additional datatypes. All we have to do is write another instance of the trait *Encode*. However, each time we add a new datatype whose values we want to encode, we need to supply another implementation of *encode*.

In contrast, in the last chapter we have seen how generic functions allow a single definition that works for a large family of datatypes. Comparing our ad-hoc definition with the generic function for

binary encoding (the trait *MySerialize* in Annexe E), we can see that the case for lists is subsumed, in the generic definition, by three generic cases for unit, sum and product types. By viewing all datatypes in a uniform way, these three cases are sufficient to call the encoder on lists, tuples, trees, and several more complex data structures – a new implementation of *encode* is not required!

Nonetheless, there are situations in which a specific case for a specific data type – called an *ad-hoc case* – is desirable. For example, lists can be encoded more efficiently than shown above: instead of encoding each constructor, we can encode the length of the list followed by encodings of the elements. Or, suppose that sets are represented as trees; the same set can be represented by multiple trees, so a generic equality function should not compare sets structurally, and therefore we need an ad-hoc case for sets.

Defining ad-hoc cases for ad-hoc polymorphic functions is trivial: we just add a new instance of *Encode* with the desired implementation. For the generic version of the binary encoder, the addition of a new case is, however, very difficult. This is one instance of the expression problem (Wadler, 1998): each case of the function definition implements a method of class *Generic*, and adding a new case later requires the modification of the class. We say that generic functions written in this style are not *extensible*, and that the approach is not *modular*, because non-extensibility precludes writing a flexible generic programming library. In a sentence, generic functions are more concise, but ad-hoc polymorphic functions are more flexible.

The specific contributions of this chapter are:

- In Section 5.2 we discuss the relationship between the expression problem and the extensibility of generic functions. Because our generic functions are implemented using visitors, a solution that solves the problem of extensibility on visitors (and the expression problem) will necessarily solve the problem of extensibility on generic functions. We believe that this is an important insight, since it allows us to investigate the application of solutions to the extensibility of visitors to generic functions and vice-versa.

- In Section 5.3, we give an encoding of extensible generic functions using internal visitors. This encoding in based on our previous work (Oliveira *et al.*, 2006) and has the advantage of being simple while allowing the addition of extensions. However, there are some issues with mutually recursive generic functions that can preclude extensibility in a different way.

- In Section 5.4 we provide a extensible generic pretty printer as an example of an extensible

generic function.

- In Section 5.5 we present a different solution for the extensibility problem of generic functions, inspired by work on the extensibility of visitors by Odersky and Zenger (2005a). This solution has the advantage that it works for any encoding and that it can avoid the extensibility issues related to mutually-recursive generic functions. We demonstrate its application to generic functions by revisiting the extensible pretty printer example.

- In Section 5.6 we compare the approaches from Sections 5.3 and 5.5. While the latter approach seems to be superior, at first glance, since it can be applied to any kind of visitor and the problem of mutually-recursive functions is not so problematic, there are few subtle advantages of the former approach. In particular, because the former approach has a single family of types, extensions do not need to be closed and they are compatible with each other.

Finally, in Section 5.7, we discuss the results of this chapter.

5.2 Generic Functions and The Expression Problem

Wadler (1998) called the need for extensibility in two dimensions (adding new variants *and* new functions) the expression problem. According to him, a solution for the problem should allow the definition of a datatype, and the addition of new variants to such a datatype as well as the addition of new functions over that datatype. A solution should not require recompilation of existing code, and it should be statically type safe: applying a function to a variant for which that function is not defined should result in a compile-time error.

In traditional OO development, extensible datatypes are easy to implement, but we can usually only have a fixed set of functions — the datatypes correspond to a hierarchical structure where all classes satisfy an interface, which defines the type of the datatype and the (fixed) set of operations that it allows. In contrast, when we use the VISITOR pattern, adding new functions is easy, but adding new variants of the datatype is much harder because the visitor interface declares the (fixed) set of variants and that is the only interface that is available to the *accept* method in the composite.

The problem of extensible generic functions (with the GM approach) fundamentally reduces to the extensibility problem in visitors. While there has been some work on extensible generic functions (Hinze and Peyton Jones, 2000; Lämmel and Peyton Jones, 2005) and even on the relation

with the expression problem (Löh and Hinze, 2006), we are not aware of any work that explicitly explores the relation with extensibility of visitors. By reducing the problem of extensibility of generic functions to the problem of extensibility of visitors we can expect that solutions for one of the problems can be applied to the other problem. This provides interesting insight; as we shall see, two solutions presented in this chapter solve the extensibility problem, but while the first one was originally developed in the context of extensibility of generic functions, the second one was inspired by a solution to the extensibility problem of visitors in Scala.

5.2.1 The Extensibility Problem of Visitors

In Section 4.4 we have seen that the implementation of GM corresponds to a visitor: the trait *Generic* (which defines the body of our generic functions) is the visitor component and the trait *Rep* is the composite component. If we would like to add additional cases to our visitor, we would naturally consider having subtypes of *Generic* with the additional cases. However, while intuitively this is the right thing to do, there are a couple of problems to solve first. Let's look at the definition of *Generic* in Figure 4.10 and recall the definition of the *accept* method in *Rep* to see where the problems are.

$$\textbf{def}\ \mathit{accept}[s <: \mathit{Strategy}, x <: \mathit{TypeConstructor}]$$
$$(v : \mathit{Generic}\ \{\textbf{type}\ X = x; \textbf{type}\ S = s\})\ (\textbf{implicit}\ \mathit{decom} : \mathit{Decompose}[s]) : x\ \{\textbf{type}\ A = T\}$$

Abstracting the visitor in the composite. The first problem (already noted) is that the *accept* method of *Rep* (the composite) refers to the specific visitor *Generic*. Although we can use a subtype of *Generic* as an argument to *accept*, in the definition of *accept* we can only (type-safely) call methods that are known to the *Generic* trait. A possible solution for this problem is to make *Rep* also parametrizable by the visitor. Interestingly, this is already possible to do using our visitor library, since the *Composite* is itself parametrizable by a visitor. So we could have used

$$\textbf{type}\ \mathit{GRep}[V <: \mathit{Generic}, T] = \mathit{Composite}[V, T]$$

instead of *Rep* $[T]$, which would allow us to refine the visitor used by the accept method with *subtypes* of *Generic*.

Abstracting the visitor in the *visit* methods. The second problem that we have is that the types of the recursive arguments in *Generic* also depend on the visitor and we cannot just use

type $Rec\,[t] = R\,[Generic, t]$ to type those arguments because this would again preclude the use of information only available to subtypes of *Generic*. So, again we basically need to be able to abstract over any subtypes of *Generic*. One way to accomplish this is by parametrizing the *visit* methods that have recursive occurrences with an extra visitor type argument. For example, the *plus* method in *Generic* could be defined as:

$$\textbf{def}\ plus[v <: Generic, a, b]\ (a : R[v, a], b : R[v, b]) : X\ \{\textbf{type}\ A = Plus[a, b]\}$$

The two problems identified here are essentially the problems that make the visitor pattern not extensible. Although we have exemplified the problems with GM, other visitors would have the similar problems. One of the reasons why this extensibility of visitors is not solved in mainstream languages is basically because those languages lack the proper (type) abstractions to encode possible solutions. In other settings (which include Scala) various solutions for the problem have been proposed (Odersky and Zenger, 2005a; Ernst, 2004; Nystrom *et al.*, 2004). Our visitor library already supports extensible visitors (since both *Composite* and *R* are already visitor-parametrized); however, we need to implement concrete visitors differently to account for this extra parametrization.

5.3 Extensibility in Internal Visitors

In the previous section we identified two problems that made the Visitor pattern hard to extend. One of the problems is that, in general, we need to abstract the visitor in the *visit* methods because recursive occurrences may depend on the concrete visitor. However, when we use internal visitors this problem does not exist because recursive occurrences of internal visitors do not depend on the concrete visitor. Making use of this observation we will, in this section, show that it is possible to develop internal visitors that can be made extensible in a simple way. This work is inspired by previous work done by the author with Hinze and Löh (Oliveira *et al.*, 2006).

5.3.1 Simple Extensible Visitors

Our visitor library allows us to define visitors that are strategy-generic, however it is also possible to define strategy-specific visitors. In Figure 5.2 we can see how to encode an internal version of the visitor *Generic* using our visitor library. Here we opted to specialize the types of the recursive

```
trait Generic extends Visitor {
  type S = Internal
  def unit  : X {type A = One}
  def int   : X {type A = int}
  def char : X {type A = char}
  def plus[a, b] (a : X {type A = a}, b : X {type A = b}) : X {type A = Plus[a, b]}
  def prod[a, b] (a : X {type A = a}, b : X {type A = b}) : X {type A = Prod[a, b]}
  def view[a, b] (iso : Iso[b, a], a: ⇒ X {type A = a}) : X {type A = b}
}
```

Figure 5.2: An internal version for *Generic* using the visitor library.

references to the form $X\ \{\textbf{type}\ A = \alpha\}$ instead of using $R[v, \alpha]$ directly, because if we had used the latter form we would need to do some extra conversions, which would affect the readability of the code. We can justify the use of the former form by observing that when $S = Internal$ the type Y (which represents the type of recursive references in the trait *Internal*) is set to $X\ \{\textbf{type}\ A = \alpha\}$ and, therefore, we can just use that type directly. Note that this version of *Generic* is equivalent to the one presented in Figure 4.3.

5.3.2 Extending Generic Functions with Extra Cases

Adding extra meta-information to generic functions The generic serializer/deserializer functions in Chapter 4 are two examples of generic functions that could be defined just using the structure of visitors. However, for some other generic functions additional meta-information may be required. For example, a pretty printer may want to display information such as the name of the actual class associated with the object in the pretty printed string. While information such as this one is often available through mechanisms like reflection, we do not need to rely on the existence of these mechanisms (or we may just not want to rely on them). We can add that meta-information to a generic function by extending *Generic* with an extra method wrapping up the desired information.

```
trait GenericConstr extends Generic {
  def constr[a] (name : String, arity : int, g : X {type A = a}) : X {type A = a} = g
}
```

The method *constr* adds the constructor name and its arity to a generic function and is given a default definition that basically ignores the extra meta-information, but can be overridden later. Generic functions that require meta-information about constructors can be defined using *GenericConstr*

instead of *Generic*. In Section 5.4 we will see an example of a generic function that uses the extra meta-information in its definition.

Extending generic functions with ad-hoc cases The main motivation for extensibility of generic functions comes from the fact that, sometimes, we want to override the generic functionality for a particular datatype at a particular generic function. For example, suppose that we want to use a different encoding of lists than the one derived generically: a list can be encoded by encoding its length, followed by the encodings of all the list elements. For long lists, this encoding is more efficient than to separate any two successive elements of the lists and to mark the end of the list.

The trait *Generic* is the base class of all generic functions, and its methods are limited. If we want to design a generic programming library, it is mandatory that we constrain ourselves to a limited set of frequently used types. Still, we might hope to add an extra case by extending *Generic*:

```
trait GenericList extends GenericConstr {
  def list[a] (a : X {type A = a}) : X {type A = List[a]} =
    view (listIso, plus (constr ("Nil", 0, unit), constr ("Cons", 2, prod (a, list[a] (a)))))
}
```

This declaration introduces a trait *GenericList* as a subclass of *Generic*. The subclass contains a single method *list*. By default, *list* is defined in a similar way to the method *listRep* (from Section 4.3.2) — except that here we already included extra meta-information. However, in contrast to *listRep*, the default definition of *list* can be overridden (in the instances of *GenericList*), and consequently we can change the default behaviour of a particular generic function for lists. For example, here is how to define the more efficient length-prefixed encoding for lists:

```
trait EncodeList extends GenericList {
  override type X = Serialize
  override def list[a] (a : X {type A = a}) =
    new Serialize {
      type A = List[a]
      def serialize (x : List[a]) : String =
        int.serialize (length (x)) + concatMap ((y : a) ⇒ a.serialize (y), x)
    }
}
implicit object encodeList extends MySerialize with EncodeList
```

The trait *EncodeList* extends the *GenericList* and overrides the case for lists in the generic function. Using Scala's mixin composition, we can now create an object *encodeList*, by combining

```
implicit def RUnit = new GRep[Generic, One] {
  def accept[s <: Strategy, x <: TypeConstructor]
    (vis : Generic {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
      vis.unit
}
implicit def RInt = new GRep[Generic, int] {
  def accept[s <: Strategy, x <: TypeConstructor]
    (vis : Generic {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
      vis.int
}
implicit def RChar = new GRep[Generic, char] {
  def accept[s <: Strategy, x <: TypeConstructor]
    (vis : Generic {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
      vis.char
}
implicit def RPlus[v <: Generic, a, b] (a : GRep[v, a], b : GRep[v, b]) = new GRep[v, Plus[a, b]] {
  def accept[s <: Strategy, x <: TypeConstructor]
    (vis : v {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
      vis.plus (a.accept (vis), b.accept (vis))
}
implicit def RProd[v <: Generic, a, b] (a : GRep[v, a], b : GRep[v, b]) = new GRep[v, Prod[a, b]] {
  def accept[s <: Strategy, x <: TypeConstructor]
    (vis : v {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
      vis.prod (a.accept (vis), b.accept (vis))
}
```

Figure 5.3: A less ad-hoc dispatcher.

the generic serialization function *MySerialize* with the ad-hoc case *EncodeList*.

5.3.3 Extensible Representations

By using internal visitors instead of strategy generic visitors, we have the advantage that there is no problem of abstracting visitors in the *visit* methods (since there are no visitors to abstract from in the first place), which was one of the problems that we identified as precluding extensibility of visitors. We are then left with the problem of abstracting the visitors in the composites. But, as we have argued in Section 5.2, this can be done if we use *GRep* instead of *Rep*. In Figure 5.3 we see how to use *GRep* to define extensible representations. The instances of *GRep* are very similar to the corresponding *Rep* instances, except that they are now parametrized by the visitor and, for

recursive representations, we use the *accept* method directly instead of going via *decompose*. The structural cases *Unit*, *Plus* and *Prod* together with the base cases *int* and *char* are all handled in *Generic*, and therefore we only need *Generic* as the visitor type parameter. However, for *List*s the visitor parameter must be constrained differently: we need to use *GenericList* instead, since this is where the *List* case is handled.

```
implicit def RList[v <: GenericList, a] (a : GRep[v, a]) = new GRep[v, List[a]] {
  def accept[s <: Strategy, x <: TypeConstructor]
    (vis : v {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
      vis.list (a.accept (vis))
}
```

Had we used *Rep* instead of *GRep*, we would have not been able to define a representation for lists, since we could not refine *Generic* to *GenericList*.

We can define an extensible generic serialization function by specializing the argument *X* in the visitor to *Serialize* and using *GRep* for the representations:

```
def extSerial[v <: Generic, t] (x : t) (r : GRep[v, t]) (implicit vis : v {type X = Serialize}) =
  r.accept (vis).serialize (x)
```

This approach is extensible, modular and type-safe and allows us to write a very flexible generic programming library.

5.4 Example: An Extensible Generic Pretty Printer

In this section we present a practical example where extensibility plays a crucial role in the definition of a generic function. The generic function in question is an *extensible generic pretty printer*: an example based on the non-modular version presented in Hinze (2004) (which was itself inspired by Wadler (2003)).

5.4.1 A Generic Pretty Printer

Mainstream object-oriented programming languages such as Java or C# define a method *toString* () in the top-most class of their class hierarchy. The intention of this method is to provide a human-readable string representation of the corresponding object. This can be useful for many purposes and it is advisable that programmers implement this method in their classes. However, this can be

```
trait PPrint extends TypeConstructor {
  def pprint (x : A) : Document
}
trait GenericPrint extends GenericConstr {
  type X = PPrint
  def unit = new PPrint {type A = One; def pprint (x : A) = empty}
  def int = new PPrint {type A = int; def pprint (x : A) = text (x.toString ())}
  def char = new PPrint {type A = char; def pprint (x : A) = text (x.toString ())}
  def plus[a, b] (a : X {type A = a}, b : X {type A = b}) = new PPrint {
      type A = Plus[a, b]
      def pprint (x : A) = x.accept (new PlusVisitor[a, b, Document] {
        def inl (y : a) = a.pprint (y)
        def inr (z : b) = b.pprint (z)
      })}
  def prod[a, b] (a : X {type A = a}, b : X {type A = b}) = new PPrint {
      type A = Prod[a, b]
      def pprint (x : A) = x.accept (new ProdVisitor[a, b, Document] {
        def prod (y : a, z : b) = a.pprint (y) ◇ break ◇ b.pprint (z)
      })}
  def view[a, b] (iso : Iso[b, a], a: ⇒ X {type A = a}) = new PPrint {
    type A = b
    def pprint (x : A) = a.pprint (iso.from (x))}
  def constr[a] (name : String, arity : int, a : X {type A = a}) = new PPrint {
    type A = a
    def s = text (name)
    def pprint (x : A) = if (arity ≡ 0) s else
      group (nest (1, (text ("(") ◇ s ◇ break ◇ a.pprint (x) ◇ text (")"))))}
}
```

Figure 5.4: A Generic Prettier Printer

tedious since much of the code required tends to be longwinded and repetitive. One alternative to this "ad-hoc" way of writing a function is to define a generic function that can be reused by specific instances, avoiding the tedious code.

Figure 5.4 presents an instance of *Generic* that defines a generic pretty printer. The pretty printer makes use of Scala's *scala.text* package, which contains pretty printing combinators. These combinators generate a value of type *Document* that can be rendered into a string afterwards. For the structural cases, the *unit* function returns an empty document; *plus* decomposes the sum and pretty prints the value in both cases; for products, we pretty print the first and second components separated by a line. For base types *char* and *int* we return a basic *Document* based on the *toString*

result for those types. The *view* case uses the isomorphism to convert between the user-defined type and its structural representation. Finally, since pretty printers require extra meta-information, the function *constr* adds that information to the resulting String.

Similarly to other generic functions, the method *pretty* provides an easy-to-use interface for the generic function. To use *pretty*, we need a value of type t together with a representation of t and a visitor where $X = PPrint$. This last argument can be implicitly passed, but the representation of t is not implicit just because Scala's type system does not allow it (since their types are not contractive).

```
def pretty[v <: Generic, t] (x : t) (r : GRep[v, t]) (implicit vis : v {type X = PPrint}) = {
  var writer = new OutputStreamWriter (System.out);
  r.accept (vis).pprint (x).format (80, writer);
  writer.flush ();
}
```

The method *pprint* (*x*) produces a *Document*, which contains a method *format* taking the number of characters that we want to have per line and an *OutputWriter* (where we write out the result produced by the pretty printer). We assume the standard 80 characters per line and write the result to the standard output.

5.4.2 Pretty Printing Trees

As a first example for the pretty printer, let's create an implementation of binary trees using our visitor library. This implementation is shown in Figure 5.5. Furthermore, we assume that there is already an isomorphism *treeIso* between trees and sums of products.

To use generic functions on trees, we proceed in the same way as we did for lists, creating a subtype of *GenericConstr* and providing an instance of *GRep* (that we omit here).

```
trait GenericTree extends GenericConstr {
  def tree[a] (a : X {type A = a}) : X {type A = Tree[a]} =
    view (treeIso, plus (constr ("Empty", 0, unit),
      constr ("Fork", 3, prod (a, prod (tree[a] (a), tree[a] (a))))))}
```

Providing a pretty printer that supports ad-hoc cases for *Tree* amounts to declaring an object that uses mixin composition to inherit from both *GenericPrint* and *GenericTree*. However, in this case we do not override the default behaviour of the generic pretty printer because, for trees, it will already produce the desired result.

```
implicit object prettyTree extends GenericPrint with GenericTree
```

```
trait TreeVisitor[a] extends Visitor {
  type Rec = R[TreeVisitor[a]]
  def empty : X
  def fork (x : a, l : Rec, r : Rec) : X
}
type Tree[a] = Composite[TreeVisitor[a]]
def Empty[a] = new Tree[a] {
  def accept[s <: Strategy, x]
    (vis : TreeVisitor[a] {type X = x; type S = s}) (implicit decompose : Decompose[s]) : x =
      vis.empty
}
def Fork[a] (x : a, l : Tree[a], r : Tree[a]) = new Tree[a] {
  def accept[s <: Strategy, x]
    (vis : TreeVisitor[a] {type X = x; type S = s}) (implicit decompose : Decompose[s]) : x =
      vis.fork (x, decompose.dec (vis, l), decompose.dec (vis, r))
}
abstract class VTree[s <: Strategy, a, b] (implicit decompose : Decompose[s])
  extends VisitorFunc[TreeVisitor[a], s, b] (decompose) with TreeVisitor[a]
```

Figure 5.5: A Visitor for binary trees

To see the pretty printer in action, let's consider a small function that produces trees with some integer value on top, the predecessor on the nodes below and so on, stopping when the last layer of nodes has value 1 (for each node):

```
def genTree (x : int) : Tree[int] =
  if (x ≡ 0) Empty[int] else Fork (x, genTree (x − 1), genTree (x − 1))
```

The call *pretty* (*genTree* (4)) (*RTree* (*RInt*)) would produce the following output:

```
(Fork
 4
 (Fork
    3
    (Fork 2 (Fork 1 Empty Empty) (Fork 1 Empty Empty))
    (Fork 2 (Fork 1 Empty Empty) (Fork 1 Empty Empty)))
 (Fork
    3
    (Fork 2 (Fork 1 Empty Empty) (Fork 1 Empty Empty))
    (Fork 2 (Fork 1 Empty Empty) (Fork 1 Empty Empty))))
```

As we can see, the pretty printer breaks the result into multiple lines, since a single line would

```
def printList[a] (x : List[a]) (g : PPrint {type A = a}) : Document =
  x.accept[External, Document] (new VList[External, a, Document] {
    def rest (l : List[a]) : Document =
      l.accept[Internal, Document] (new VList[Internal, a, Document] {
        def nil = text ("]")
        def cons (y : a, ys : Rec) = text (",") ◇ break ◇ g.pprint (y) ◇ ys.get
      })
    def nil = text ("[]")
    def cons (y : a, ys : Rec) = group (nest (1, text ("[") ◇ g.pprint (y) ◇ rest (ys.get)))
  })
object prettyList2 extends GenericPrint with GenericList {
  override def list[a] (a : PPrint {type A = a}) = new PPrint {
    type A = List[a]
    def pprint (x : List[a]) : Document = printList (x) (a)
  }
}
```

Figure 5.6: Ad-hoc pretty printing for lists.

take more than 80 characters. Each level of the tree is nicely indented, and it is clear from the layout how the tree is structured.

5.4.3 Pretty Printing Lists

For most user-defined types like *Tree*, our generic pretty printer works nicely. However, for some types such as lists or sets, the default generic pretty printing algorithm may not be the most appropriate. The problem with lists is that we normally use a special mix-fix notation to represent them: instead of writing "*Cons* $(3, \mathit{Cons}\ (2, \mathit{Nil}))$" we usually write something like "$[3, 2]$". If we just use the default behaviour that is given by *GenericList* (see Section 5.3.2) we would get the more longwinded notation for lists. In order to adapt our generic function in such a way that the short notation is used we need to override the default functionality by providing an ad-hoc definition.

We show the code for pretty printing lists in Figure 5.6. The method *printList* is used by the overridden definition of *list* (in *prettyList2*) to provide the short notation. It is important to note that this ad-hoc extension to the generic function is modular: we do not need to modify the module where the main body of the generic function is.

Pretty Printing Lists of Characters

In many programming languages, strings are represented as lists of characters. The notation for strings differs from the conventional list notation: instead of "`['H','e','l','l','o']`" we usually have "`"Hello"`". As we shall see, it is possible to modify our pretty printer so that it also supports this notation. However, this modification is not entirely modular. In order to handle strings as lists of characters, not only do we need to treat lists in a special manner, but we also need to handle lists of characters diferently from lists of any other element type. We thus have to implement a *nested case analysis on types*. In order to do this nested case analysis, we need to anticipate this possibility and include a function *pprintList* in the trait *PPrint*.

```
trait PPrint extends TypeConstructor {
  def pprint (x : A) : Document
  def pprintList (x : List[A]) (g : PPrint {type A = PPrint.this.A}) : Document =
    printList (x) (g)}
```

By default, pretty printing a list of values of some type *A* will use the standard list notation. However, when *A* is the type *char* we override the definition of *pprintList* in the body of the generic function so that it handles lists of characters specially.

```
trait GenericPrint extends GenericConstr {
  ...
  def char = new PPrint {
    type A = char;
    def pprint (x : A) = text (x.toString ())
    override def pprintList (x : List[A]) (g : PPrint {type A = char}) =
      text ("\"" + (x.accept[Internal, String] (new VList[Internal, Char, String] {
        def nil = "\""
        def cons (y : char, ys : Rec) = y.toString () + ys.get})))}
  ...}
```

The final touch is to modify *prettyList2* so that it calls *pprintList*, thus forwarding *pprint* to the appropriate functionality.

```
object prettyList2 extends GenericPrint with GenericList {
  override def list[a] (a : PPrint {type A = a}) = new PPrint {
    type A = List[a]
    def pprint (x : List[a]) : Document = a.pprintList (x) (a)
  }
}
```

In contrast to supporting just the list notation (but not the string notation), this extension is not modular since we need to modify both the traits *PPrint* and *GenericPrint*. Still, it is interesting to observe that such support is possible and, arguably, when there are a few exceptional cases like strings this may be acceptable.

The problem with our implementation of the generic function supporting the string notation is that, because we are basically using a Church encoding, we are restricted to a simple pattern of recursion. Since nested case analysis does not fit this recursion scheme, we are forced to manually built in support for it.

One immediate question that arises is wheather we can use other kinds of encodings to tackle the same question modularly. For example, unlike Church encodings, Parigot encodings are very liberal in the recursion scheme that can be used. However, for other encodings, recursive references may depend on the visitor, which raises the problem of abstracting the visitor in the *visit* methods that the Church encoding does not have. In the next section we see how to use a different technique that allows extensibility for other encodings.

5.5 Extensibility on Visitors of any Kind

In the previous section we have a solution for extensibility of internal visitors. In this section we present a different solution for the extensibility problem of visitors that works for any encodings. This solution is inspired by a technique originally introduced by Odersky and Zenger (2005a). We demonstrate its application to generic functions by revisiting the extensible pretty printer example. The full code for this section is presented in Appendix H.

5.5.1 Extensibility on Generic Encodings

Internal visitors do not have recursive occurrences in the *visit* methods that depend on the visitors themselves. However, other encodings may not have this property and we need to solve this problem if we want to achieve extensibility. One solution, mentioned in Section 5.2, would be to parametrize on the type of concrete visitors in the *visit* methods. Although this could be done, the code would become cluttered with type parametrizations (both from the *visit* methods and from the Composite). Fortunately, using Scala abstract types and nested traits we can tackle this problem in a different manner, with a minimum amount of noise. The idea is that instead of abstracting

```
trait ExtensibleGMVisitor {
  type Gen <: Generic
  type GenWith[x <: TypeConstructor, s <: Strategy] = Gen {type X = x; type S = s}
  trait Generic extends Visitor {
    type Rec[t] = R[Gen, t]
    def unit  : X {type A = One}
    def int   : X {type A = int}
    def char : X {type A = char}
    def plus[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Plus[a, b]}
    def prod[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Prod[a, b]}
    def view[a, b] (iso : Iso[b, a], a: ⇒ Rec[a]) : X {type A = b}
  }
  type Rep[T] = Composite[Gen, T]
  implicit def RUnit = new Rep[One] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
        vis.unit
  }
  ...
  implicit def RProd[a, b] (implicit a : Rep[a], b : Rep[b]) = new Rep[Prod[a, b]] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
        vis.prod (decompose.dec (vis, a), decompose.dec (vis, b))
  }
  ...
}
```

Figure 5.7: A parametrized module for generic functions

the concrete visitors in each *visit* method and in each composite element, we have a trait with an abstract type over the concrete visitor in use. In Figure 5.7, we can see how to apply this solution to the *Generic* visitor.

The top-level trait *ExtensibleGMVisitor* has an abstract type *Gen*, which is a subtype of the inner trait *Generic*. The method *accept* in *Rep* uses the abstract type *Gen* to define the type of its argument *vis*, instead of using *Generic* directly. This fact means that it is up to the instances of *ExtensibleGMVisitor* to define which specific kind of *Generic* is going to be used. It may well be that such an instance would instantiate *Gen* to *Generic* itself. In that case not much would be gained. However, any subclass would also be a valid instantiation, and therefore we could make use of any extra cases defined in such subtypes. In the next two subsections we shall see how we

can independently create two extensions to generic functions.

5.5.2 Supporting Lists

The trait *ExtensibleGMList*

```
trait ExtensibleGMList extends ExtensibleGMVisitor {
  type Gen <: Generic
  trait Generic extends super.Generic {
      def list[a] (a : Rec[a]) : X {type A = List[a]}
  }
  implicit def RList[a] (implicit a : Rep[a]) : Rep[List[a]] = new Rep[List[a]] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
        vis.list[a] (decompose.dec (vis, a))
  }
}
```

extends *ExtensibleGMVisitor* and defines a new trait *Generic*, which is a subtype of the original. Because this new version of *Generic* is a subtype of the original one, we can refine *Gen* by further constraining it to be a subtype of this new version. This modification means that functions that have arguments that are instances of *Gen* can now assume any new methods declared in the new version of *Generic*. In particular, we have extended the original interface with a new method *list* for handling the case for lists and we have defined a representation *RList* making use of that method.

5.5.3 Supporting Meta-Information and Pretty Printing

In Section 5.3.2 we presented an extension of *Generic* that supported constructor information. We shall do the same now and use that added functionality to define a generic pretty printing function using our new encoding. The trait *ExtensibleGMConstr* defines an extension of *ExtensibleGMVisitor* supporting extra meta-information.

```
trait ExtensibleGMConstr extends ExtensibleGMVisitor {
  type Gen <: Generic
  trait Generic extends super.Generic {
    def constr[a] (name : String, arity : int, g : Rec[a]) : X {type A = a}
  }
  def RConstr[a] (name : String, arity : int, a : Rep[a]) : Rep[a] = new Rep[a] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s]) =
```

```
        vis.constr (name, arity, decompose.dec (vis, a))
    }
    ...
```

As in Section 5.5.2, we extend the inner trait *Generic* with a new method (the method *constr*). This method supports meta-information, and can be used to define generic functions that require such support.

Also in *ExtensibleGMConstr*, we define a new generic function for pretty printing. The code for this generic function can be found in Appendix H. The trait *PPrint* defines the signature of the generic function; the trait *GenericPrint* defines the main body of the generic function; and, finally, the method *pretty* provides a simple interface for pretty printing. A noticeable difference to the solution using internal visitors in Section 5.4.1 is that, because the type *Gen* is abstract, we do not know which concrete *Generic* visitor is going to be used; therefore, we need to require that the self type of *GenericPrint* is *GenWith*[*PPrint*, *External*]. Another difference, is that, since we opted to use an external visitor in the definition of the pretty printing functions, we need to explicitly make recursive calls (by using the *accept* method).

5.5.4 Merging List and Constructor Support

After defining extensions supporting lists and constructors we can now merge both functionalities together. To do that we create a new trait that, using mixin composition, inherits from both *ExtensibleGMConstr* and *ExtensibleGMList*. For the inner trait *Generic* we do something similar, inheriting from the two traits *Generic* in the top-level traits. We also define a new trait *GenericPrint*, which inherits its functionality from the traits *GenericPrint* in *ParigotConstr* and the newly defined *Generic*, and overrides the default case for lists adding support to the mixfix bracket notation. The code is shown in Figure 5.8.

As a remark we should note that certain programming languages support *deep mixin composition* (Aracic *et al.*, 2006; Ernst, 1999), which could be used to remove some of the burden from the programmer when mixing in related extensions. With deep mixin composition, we could automatically mixin inner traits. In Scala we need to do this composition manually.

```
trait ExtensibleGMListConstr extends ExtensibleGMConstr with ExtensibleGMList {
  type Gen <: Generic
  trait Generic extends super[ExtensibleGMConstr].Generic
    with super[ExtensibleGMList].Generic
  trait GenericPrint requires GenWith[PPrint, External]
    extends super.GenericPrint with Generic {
      override def list[a] (a : Rec[a]) : PPrint {type A = List[a]} = new PPrint {
        type A = List[a]
        def pprint (x : A) = pprintl (x) (a.get, GenericPrint.this)
      }
  }
  def pprintl[a] (x : List[a]) (implicit a : Rep[a], v : GenWith[PPrint, External]) =
    x.accept[E, Document] (new VList[E, a, Document] {
      def rest (l : List[a]) : Document =
        l.accept[I, Document] (new VList[I, a, Document] {
          def nil = text ("]")
          def cons (y : a, ys : Rec) = text (",") :: break :: a.accept (v).pprint (y) :: ys.get
        })
      def nil = text ("[]")
      def cons (y : a, ys : Rec) = group (nest (1, text ("[") :: a.accept (v).pprint (y) ::
                                          rest (ys.get)))
    })
}
```

Figure 5.8: Merging Support for Constructor and Lists

5.5.5 Creating a New Module

We can create a new object that wraps up all wanted functionality and closes the door to extensibility by fixing the abstract type *Gen*. In our example we inherit from the trait *ExtensibleGMListConstr* and set the abstract type *Gen* to the type *Generic* from that trait. Since *ExtensibleGMListConstr* already supports lists and meta-information in generic functions as well as a generic pretty printing function, we can create a function *myTest* that uses the pretty printing function to print a list using a mix-fix notation. This new module could now be imported from other Scala code and be used as any other Scala library.

```
object testExtensibleGMListConstr extends ExtensibleGMListConstr {
  type Gen = Generic
  implicit object prettyPrint extends GenericPrint
  def listTest = Cons ('3', Cons ('2', Cons ('1', Nil[char])))
  def myTest = pretty (listTest) (RList (RChar), prettyPrint)}
```

5.5.6 Supporting String Notation

Finally, and to complete the pretty printer example using external visitors, we revisit the problem of supporting the string notation. The solution presented here follows closely Hinze's own solution for the problem. However, in contrast to his solution, our solution is extensible and modular.

As a first step, we create a subtype of *Generic* that implements a common default functionality for all cases of a generic function. This default functionality is provided by the method *deflt*, that takes a representation of some type *a* and gives an instance of the generic function for that type. This trait could be incorporated, for example, in *ExtensibleGMVisitor*.

```
trait GenericDefault extends Generic {
  type S = External
  def deflt[a] (a : Rep[a]) : X {type A = a}
  def unit  = deflt (RUnit)
  def int   = deflt (RInt)
  def char = deflt (RChar)
  def plus[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Plus[a, b]} =
    deflt (RPlus (a.get, b.get))
  def prod[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Prod[a, b]} =
    deflt (RProd (a.get, b.get))
}
```

Figure 5.9 shows the code for supporting the string notation. Using *ExtensibleGMListConstr* as a base for the trait *ExtensibleGMString*, we define a generic function that handles pretty printing of list values playing the same role as *pprintList* in *PPrint* did on the internal visitor solution. The body of that function is defined on the trait *GenericPrintList*, which extends *GenericDefault* and implements *deflt* so that it calls the function *prettyDoc* that is defined in *ExtensibleGMConstr*. We also override *char* in *GenericPrintList* so that it uses the string notation to print a list of characters. Then, we define a method *prettyListDoc* that provides a convenient way to call the generic function. Finally, we update the *list* method in *GenericPrint*, making it call *pprintl*.

To summarize, this solution corresponds, in essence, to the definition of two mutually recursive (generic) functions *GenericPrint* and *GenericPrintList*, in which the case for lists in *GenericPrint* is handled by *GenericPrintList* that, in turn, handles all the cases except characters by calling back *GenericPrint*. Unfortunately, the code in Figure 5.9 did not compile in the version of Scala that we tried, although we believe it should. The problem is discussed briefly in Appendix H and a

```
trait ExtensibleGMString extends ExtensibleGMListConstr {
  trait PPrintList extends TypeConstructor {def pprintList (x : List[A]) : Document}
  implicit def defPrint : GenWith[PPrint, External]
  trait GenericPrintList requires (GenWith[PPrintList, External] with GenericDefault)
                          extends GenericDefault {
    type X = PPrintList
    def deflt[a] (a : Rep[a]) = new PPrintList {type A = a;
        def pprintList (x : List[A]) = prettyDoc (x) (RList (a), defPrint)}
    override def char = new PPrintList {
      type A = char
      def pprintList (x : List[char]) =
        text ("\"" + (x.accept[Internal, String] (new VList[Internal, Char, String] {
          def nil                    = "\""
          def cons (y : A, ys : Rec) = y.toString () + ys
        })))
    }
    def view[a, b] (iso : Iso[b, a], a: ⇒ Rep[a]) = new PPrintList {
      type A = b
      def pprintList (x : List[A]) : Document =
        pprintl (
          x.accept[Internal, List[a]] (new VList[Internal, A, List[a]] {
            def nil = Nil[a]
            def cons (x : A, xs : Rec) = Cons (iso.from (x), xs.get)
        })) (a, defPrint)
    }
    def list[a] (a : Rep[a]) = deflt (RList (a))
  }
  def prettyListDoc[t] (x : List[t]) (implicit r : Rep[t], v : GenWith[PPrintList, External]) =
    r.accept (v).pprintList (x)
  trait GenericPrint requires GenWith[PPrint, External] extends super.GenericPrint {
    override def list[a] (a : Rec[a]) : PPrint {type A = List[a]} = new PPrint {
      type A = List[a]
      def pprint (x : A) = pprintl (x) (a.get, GenericPrint.this)}
    override def constr[a] (name : String, arity : int, a : Rep[a]) = new PPrint {
      type A = a
      def s = text (name)
      def pprint (x : A) = if (arity ≡ 0) s else
        group (nest (1, (text ("(") :: s :: break ::
        a.rep (GenericPrint.this).pprint (x) :: text (")"))))}
  }
}
```

Figure 5.9: Support the string notation with the Parigot encoding.

workaround is provided.

5.6 Comparing the Two Approaches to Extensibility

The two approaches to extensibility presented in this chapter have different advantages and disadvantages. While it may be tempting to think that the solution presented in Section 5.5 is superior because it supports any kind of visitors and does not have issues with mutually recursive functions, the fact is that the solution in Section 5.3 has some interesting (subtle) advantages worth remarking upon:

- The first advantage is that there is *no need to close extensions* to be able to use functionality. With the solution in Section 5.5 we have to create an object that instantiates the abstract types, so that we can actually use the functionality that is provided (the object defined in Section 5.5.5 is an example of this). In contrast, the solution using internal visitors does not need the creation of these objects, which makes the approach simpler to use.

- The second advantage is that *there is a single family of types*. With the solution in Section 5.5, different *closed* extensions are incompatible with each other, because each closed extension has a different family of types associated with it. For example, suppose we had an object that inherited from the trait *ExtensibleGMString* and closed the extension. Then that object would define its own family of representation types *RUnit*, *RChar*, *RInt*, and so on, incompatible with those of some other closed extension. If we had several closed extensions in the same project and we did not plan carefully their usage, it may be impossible to mix the functionality of two closed extensions. In order to combine the two, we either have to modify existing code, or if we cannot access the source, we may end up duplicating code and creating a third (closed) extension. This does not happen with the solution with internal visitors, because there is one common family of types.

- The third and final advantage is that *we can easily have functions that work on different type universes (or domains)*. Suppose that we want to use our generic programming library with structures that contain functions. Not all generic functions can be sensibly applied to functions, for example, it is not (trivially) possible to serialize and deserialize those values. However, writing other generic functions for functional types may be possible and desirable.

For example we could just print the string "<< *function* >>" for functions with our generic pretty printer, and have all other values nicely printed as usual. Ideally, we would like to have serialization/deserialization functions that cannot be applied to functional values (this is, they would produce a type-error), but we would like our pretty printer to handle those values. To achieve this with the solution in Section 5.5, we would need to define two different closed extensions: one that supports functional values but has no serialization support; and one that does not support functional values but has serialization support. So, with this solution, we need to carefully plan both the generic programming library and its closed extensions. With the solution using internal visitors this is much simpler: if we do not want to support functional values for serialization we simple do not provide any objects of type *GenericFunc* {**type** X = *Serialize*} (here *GenericFunc* is a hypothetical subtype of *Generic* supporting a case for functional values).

In summary, the approach presented in Section 5.5 is more powerful because it allows us to use any kind of visitors without limitation; however, for maximum flexibility, careful planning is needed. In particular, extensions should be very finely grained, separating each different aspect, so that closed extensions can pick just the functionality they want. In contrast, the solution in Section 5.3 requires much less planning due to the existence of a single family of types, which means that all extensions can remain open and compatible with each other.

5.7 Discussion

A lot of work (Bird *et al.*, 1996; Jansson, 2000; Hinze, 2000; Löh, 2004) provides a very strong basis for generic programming, but only considers non-extensible generic functions. It was realized by several authors (Hinze and Peyton Jones, 2000; Hinze, 2004; Lämmel and Peyton Jones, 2005) that this was a severe limitation. The problem of extensibility of generic functions is related to the expression problem. In this chapter we have shown that, when we consider the GM approach, we can reduce extensibility of generic functions to the problem of extensibility on visitors. This relation means that solutions from both domains can (possibly) be applied interchangeably. The two solutions presented here can both be applied to either domain, yet each was were initially proposed as a solution for one particular domain.

The first solution that we have presented (in Section 5.3) works only for internal visitors. How-

ever, this solution is relatively simple to use and easy to implement because, with internal visitors, the *visit* methods do not have recursive arguments. Interestingly, we believe this is the first time that a solution for the extensibility of visitors using a (functional) internal variation of the pattern is presented. Most solutions that we know of have mutually dependent visitor and composite interfaces, which are closer to external visitors and are harder to extend. Alexandrescu (2001) proposed *acyclic visitors* to avoid the problem of mutual dependencies and, at the same time, allow extensibility. However, dynamic casts are required for this solution to work. Several other solutions (Odersky and Zenger, 2005a; Ernst, 2004; Nystrom *et al.*, 2004) relax the problem of mutual dependencies by parametrizing on the type of the concrete visitor, which allows extensions to refine that same type. Our second solution, inspired by Odersky and Zenger (2005a), works that way.

One problem with internal visitors is that, because the visitor has no control of the traversal, we have a limited recursion scheme available. Because of this, mutually recursive definitions (which can be used to implement nested case analysis and binary methods) are harder to implement in a modular way: we need to encode the mutually recursive definitions using a single visitor, which is normally implemented by tupling the definitions together in a class. However, this solution is non-modular, because we need to modify previously existing code. A possible way around this is to use a mixed solution that combines the two solutions that we have presented. The idea is that we modify the solution with internal visitors so that it abstracts over the hard references to the generic function in use. For example, if we are defining a generic pretty printing function we may want to abstract over *PPrint* — the trait with the type of the generic function, where we may tuple mutual definitions together. Then, if we wanted to support the string notation in a later extension, we could just refine the abstract type to a more concrete subtype of *PPrint* (with the extra definitions).

By choosing a more liberal kind of visitor, the solution using abstract types can avoid the issue with mutually-recursive visitors, meaning that nested case analysis or binary methods can be implemented without endangering modularity. In the context of generic programming, this approach provides a novel solution to the problem of extensibility of generic functions, since nearly all approaches that we know of (Lämmel and Peyton Jones, 2005; Weirich, 2006; Oliveira *et al.*, 2006) have issues with mutually recursive generic functions. The exception is a solution by Löh and Hinze (2006) that avoids this issue at the cost of type-safety: a call to an undefined case on a generic function causes a run-time error instead of a compile-time error. However, we believe our solution is preferable, because we keep all the expressive power and we do not need to abandon

type-safety.

Chapter 6

Conclusion

6.1 Summary and Contributions

In this dissertation we have argued that, with expressive type systems such as the one in Scala, we can capture many design patterns as software components. To give credibility to our thesis, we have presented a Scala implementation of a generic library for the Visitor pattern that is parametrizable over several aspects (or alternative implementations) of the pattern. We have also shown that, because visitors are, in essence, encodings of datatypes, we can naturally apply well-known DGP techniques to our visitors. This means that we can write generic and type-safe functions that work for a very large family of visitors and thus avoid repetition of similar and tedious to write code for each concrete visitor. Finally, we have looked into the problem of extensibility, and have shown the connection between extensibility of generic functions and extensibility of visitors, investigating different solutions for both problems.

The specific contributions of this thesis are as follows:

- We have shown that an OO language supporting generics and abstract types can be used to capture different aspects of the Visitor pattern as a software component. In particular, as well as DGP, we have shown that visitors can be *strategy-generic*, avoiding the design choice in the traditional presentation of the Visitor pattern of who controls the traversal. In other words, the control of the traversal (the strategy) is parametrizable and the programmer only needs to commit to a particular traversal strategy when defining a new function (using visitors) instead of hard-wiring a particular strategy on the implementation of the visitor.

- We have implemented a visitor notation, directly in Scala, that allows visitors to be treated as functions that take composites as parameters. With this notation, functions defined using visitors look like functions defined by (a simple form of) pattern matching, which is more intuitive for programmers.

- We have adapted the GM approach to DGP to Scala and shown how it can be used in combination with our visitor library to define generic functions over those visitors. Furthermore, we have exploited inheritance in Scala to provide an easy way to reuse code from other generic functions. This particular kind of reuse is valuable, but it is harder to implement in programming languages like Haskell. Finally, we have also demonstrated that we can define views to other families of visitors using GM. Specifically, we have shown how to define the so-called fixpoint view.

- We have shown how to adapt our visitor library so that it supports the definition of a particular class of indexed datatypes (which we called unnested GADTs). Although Johann and Ghani (2007) have shown how to define a family of Church encodings for nested datatypes (which are a form of indexed types not handled by our own family), we believe this is the first time that a family of datatype encodings for a form of GADTs is given.

- We have established a relationship between visitors and generic functions by showing that the essence of the GM approach is, itself, an instance of the Visitor pattern.

- This relationship between GM and visitors means that the problem of extensibility of generic functions can be reduced to the problem of extensibility of visitors. This provides new insight and leads to new solutions for extensibility problems. In particular, we show a new solution to both problems, using internal visitors, that has the advantage of simplicity. We also show a new solution to the problem of extensibility of generic functions inspired by an existing solution to the problem of extensibility of visitors. This solution has the main advantage that it works for any encoding.

- Finally, we have also seen two practical applications of generic programming in an OO language. The first application shows how to define a serialization (and deserialization) component that works, generically, for a large family of visitors. The second application is a flexible generic pretty printing component that can be adapted, if the generic behaviour of the pretty

printer is not desirable, with an ad-hoc implementation. We should emphasize that current solutions (in mainstream OO languages) for those problems tend to be inflexible, inelegant and (sometimes) type-unsafe. We believe that DGP provides a much better solution for these problems.

6.1.1 Some Extra Insights

The aspect of "*who is responsible for traversing the object structure*" (Gamma *et al.*, 1995) in the VISITOR pattern, which (as far as we know) has always been presented as non-parametrizable, can be precisely related with encodings of datatypes and with different recursion patterns. These connections reveal interesting insights, that are worth mentioning:

- The connection with encodings of datatypes provides, for example, an alternative implementation for internal visitors where the visitor and the composite are not mutually dependent (or cyclic). Most implementations of internal visitors (i.e. visitors where the visitor component has no control over the traversal) on the literature use mutually dependent visitors and composites, which makes extensibility harder to realise.

- The connection with recursion patterns provides insight into the expressive power of different implementations of visitors. For instance, internal visitors are associated with iteration offering a simple, but limited, pattern of recursion. Many of the problems often found in the visitor pattern, such as the difficulty of doing binary methods, are related to the pattern of recursion in use: it is hard to implement binary methods or nested case analysis with internal visitors, but with some other kinds of visitors (for example, external visitors), these are not problematic.

- Another insight gained from the connection with recursion patterns reveals alternative implementations of the VISITOR pattern that have not been explored in the past. While internal visitors are related to catamorphisms and external visitors correspond, basically, to case analysis, there are other recursion patterns that give rise to different implementations of visitors. For example paramorphisms, which capture primitive recursion, have a corresponding visitor implementation. The paramorphic visitors in Section 3.8.4 present an implementation of visitors of that kind.

- Finally, recursion patterns are *first-class*: we can have a function that is parametrized by a strategy, and different instantiations of that parameter will determine which recursion pattern is going to be used. With the traditional, higher-order, presentation of recursion patterns, it is not straightforward to have a function that abstracts from the recursion pattern in use, because different recursion patterns have (slightly) different types which are difficult to abstract from. Although this fact is interesting to observe, we have not yet explored any possible uses.

6.2 A Type-Theoretic Perspective on this Thesis

The main focus of this dissertation was to show how it is possible to capture the Visitor design pattern as a software component. For doing so, we were inspired by type-theoretic results about encodings of datatypes. However, the development of our visitor library lead to some generalizations of encodings of datatypes that seem to be interesting from a type-theoretic perspective. We review our results here with that perspective in mind.

Background The traditional presentation of encodings of datatypes in System F (and common variants) is of the form $T \equiv \forall X.\ (F\ R \Rightarrow X) \Rightarrow X$. In this form a datatype T can be defined by instantiating F to some sum-of-product functor. Buchlovsky and Thielecke (2005) show that an isomorphic variation of these encodings, of the form $T \equiv \forall X.\ (\prod_i F_i\ R \Rightarrow X) \Rightarrow X$, can be precisely related to the Visitor pattern. With this variation a new datatype T can be defined by providing a product of functions $V\ R\ X \equiv \prod_i F_i\ R \Rightarrow X$ (the visitor), where each function $F_i\ R \Rightarrow X$ corresponds to a *visit* method and $F_i\ R$ corresponds to the arguments of the constructor. Church and Parigot encodings (corresponding, respectively, to internal and external visitors) follow from two specific instantiations of R ($R \equiv X$ and $R \equiv \mu_{Parigot}\ V$):

$$\mu_{Church}\ V \equiv \forall X.\ V\ X\ X \Rightarrow X$$
$$\mu_{Parigot}\ V \equiv \forall X.\ V\ (\mu_{Parigot}\ V)\ X \Rightarrow X$$

Generic encodings of datatypes Although Church and Parigot encodings both follow from the same template, we were not aware of any more general abstraction that could (linguistically) capture both encodings as a particular instance of that same abstraction. In this dissertation we show that such an abstraction exists and that it can be used to define a generic encoding of datatypes:

$$\mu\ V \equiv \overbrace{\forall X\ S.\ Decompose\ S \Rightarrow \underbrace{V\ (S\ V\ X)\ X}_{Visitor} \Rightarrow X}^{accept\ method}$$

The type parameter S (called the decomposition strategy) determines which encoding is used when visiting a value of type $\mu\ V$. We can recover Church or Parigot encodings by instantiating S with either *Church* or *Parigot*, where

$$Church\ V\ X \equiv X$$
$$Parigot\ V\ X \equiv \mu\ V$$

Decomposition function and recursion patterns The type *Decompose S* defines a function

$$dec \in V\ (S\ V\ X)\ X \Rightarrow \mu\ V \Rightarrow S\ V\ X$$

whose type is closely related to the types of well-known recursion patterns. For example, when we instantiate $V\ R\ X \equiv F\ R \Rightarrow X$ (where F is some sum of products) and $S \equiv Church$ we obtain the type (modulo isomorphisms):

$$cata \in (F\ X \Rightarrow X) \Rightarrow \mu\ F \Rightarrow X$$

If we would have instantiated $S \equiv Para$ instead, we would have obtained a function

$$paraDec \in (F\ (\mu\ F, X) \Rightarrow X) \Rightarrow \mu\ F \Rightarrow (\mu\ F, X)$$

which is a 'cousin' of the paramorphism recursion pattern. The connection between *dec* and recursion patterns reveals interesting properties. For example, the '*free theorem*' (Wadler, 1989) of *cata* gives us a fusion law. The function *dec*, being a generalization of *cata*, has also a, more general, free theorem that can be specialized to obtain the fusion laws for *cata* and other recursion patterns.

Encodings of indexed datatypes In this thesis we have also shown how to encode a form of indexed types by slightly generalizing the type $\mu\ V$

$$\mu\ V\ T \equiv \overbrace{\forall X\ S.\ Decompose\ S \Rightarrow \underbrace{V\ (S\ V\ X)\ X}_{Visitor} \Rightarrow X\ T}^{accept\ method}$$

so that it takes one extra parameter T, which is the index type. Capturing families of GADTs in this way is, as far as we know, a novel result. Worth noting is the fact that, by using the variation

based on products of functions instead of one based on sums of products (i.e. of the form $(F\ U \Rightarrow X\ T) \Rightarrow X\ T$), we can specify the return types of the constructors of our indexed datatypes. With (a naive) sums of products approach it is not possible to express the dependencies between the sum types and the return type, which means that (with such an approach) we cannot express indexed types. Consequently, there is no isomorphism between sums of products and products of functions for the two variations of $\mu\ V\ T$ (that is, the variation with sums of products is less expressive than the one with products of functions). A possible way to recover the isomorphism may be to use something like $(F\ U\ T \Rightarrow X\ T) \Rightarrow X\ T$, but we have not explored this path yet.

6.3 Haskell versus Scala

We specified most parts of our library using a functional notation that is, in essence, syntactic sugar for Haskell. The option to use Haskell as a specification language had basically two motivations. Firstly, Haskell is relatively close to System F, which give us sufficient confidence that our constructions are (semantically) well-defined. Secondly, Haskell is, by far, the most widely experimentation platform for implementations of DGP.

Because Scala is significantly different from Haskell, we were curious to know if there would be any advantages (or disadvantages) of using a language like Scala instead of Haskell for the implementation of generic programming libraries. It is interesting to compare the Haskell implementation with the Scala one. We start by stating the advantages of the Haskell version:

- *Purity* - Our visitor library can strongly benefit from the theory developed around recursion patterns. In particular, we would expect that the fusion laws associated with recursion patterns could be used to optimize programs written with visitors. However, in the presence of side-effects, those laws do not hold. Because Haskell is a pure language we have the guarantee that no (hidden) side-effects exist, which makes this kind of optimization fairly straightforward to apply. In contrast, in an impure language like Scala, we do not have the same guarantees about side-effects, which makes this kind of optimizations much harder to apply.

- *Syntactical clarity* - While Scala's syntax is more elegant than Java's or C#, we found, at points, that too much syntactic verbosity was required in comparison with Haskell's equivalent. In particular, the declaration of types in Scala tends to be quite long-winded. Abstract

types especially contribute to that since, to specify the type of an abstract type in a method declaration, we need something like $t\{\textbf{type}\ A = a\}$ while in Haskell we can often achieve the same result using type-constructor polymorphism with the syntax $t\ a$. Scala has recently been extended with support for type-constructor polymorphism (Moors *et al.*, 2007), which may help on the syntactical clarity issue. Also, due to the complexity of Scala's type system, we need many more type annotations than in Haskell, which again affects clarity.

We now state the advantages of Scala:

- *Expressiveness of the type system* - The combination of subtyping, abstract types, self types, traits and mixins provides Scala with an impressively powerful type system. We found two parts of our library that are typeable in Scala but are very hard (or even impossible) to type in Haskell. The first part, not possible to type in Haskell (because it would require polymorphic kinds), is the precise type of the *Visitor* component. We discuss this in more detail in Section 6.3.1. The second part of the library that we found very hard to type in Haskell concerns the extensibility solutions. While the first solution to extensibility was inspired by the Haskell solution in EMGM, the fact is that the Haskell version feels like a workaround for the absense of proper subtyping. Specifically, the GM (non-extensible) solution would have a method $rep :: \forall g.\ Generic\ g \Rightarrow g\ t$ and the EMGM version changes the quantification of the type g to the enclosing class *Rep*, rather than the method *rep*, to allow extensibility. The Scala type for *rep* (written in a Haskell-like syntax) would be $rep :: \forall(g <: gen).\ g\ t$, where $gen <: Generic$ would be a type parameter of the enclosing class. The key difference to the Haskell version is that the type g remains universally quantified in *rep* and only the bound *gen* is quantified in the enclosing class. In Haskell it is possible to simulate abstraction over type classes (Hughes, 1999; Lämmel and Peyton Jones, 2005), which would allow us to have $rep :: \forall g.\ gen\ g \Rightarrow g\ t$. Still, this would be unsatisfactory because *gen* could be any type class (and not just a *subclass* of *Generic*). Finally, Haskell lacks a mechanism that mimics information hiding over several objects as given in Scala by the combination of nested classes and abstract types. As an analogy, to help the reader visualize a possible such mechanism in Haskell, imagine an hypothetical extension with the ability to declare type classes inside type classes combined with associated types (Chakravarty *et al.*, 2005b) that allowed type refinements of these same associated types in subclasses. The lack of such a mechanism means

that there is no straightforward way to encode the second Scala solution to the extensibility problem in Haskell.

- *Inheritance* - Another advantage of Scala is that we can easily reuse generic functions using inheritance. In Haskell, although we can simulate this form of reuse in several ways, we can not do it in a natural way.

In conclusion, the powerful type system makes Scala a very effective language for the construction of generic programming libraries — and software components in general (Odersky and Zenger, 2005b). However Scala's syntax can be verbose at times and, because Scala is not a pure functional language, reasoning about components may be hard. Haskell's advantages are the elegant syntax and the purity of the language. However, these advantages are outweighed by the limitations of the type system — specially the absence of subtyping and a mechanism that allows information hiding over several objects — which means that Haskell cannot capture (or at least not capture precisely enough) certain aspects of complex components.

6.3.1 A Slightly Inaccurate Specification

In Section 3.6.1 we defined our visitor library based on a functional specification with

$$\mathit{Composite}\ V \equiv \forall X\ S.\ \mathit{Decompose}\ S \Rightarrow \underbrace{V\ (S\ V\ X)\ X}_{\mathit{Visitor}} \Rightarrow X$$

However, as we have discussed, the visitor component did not exactly follow the specification since it should be of the form $V\ (S\ V\ X)\ X$ and yet the trait *Visitor* was defined as

```
trait Visitor {
  type X
  type S <: Strategy
  type R[v <: Visitor] = S {type X = Visitor.this.X; type V = v}
}
```

which implied a form like:

$$\mathit{Composite}\ V \equiv \mu\ (\forall X\ S.\ \mathit{Decompose}\ S \Rightarrow \underbrace{V\ S\ X}_{\mathit{Visitor}} \Rightarrow X)$$

This later form is more flexible and general because we can always recover all the instances given by the former (in the Scala code, this is done by parametrizing the type *R* with the concrete visitor that we are defining). The extra generality allows us, for example, to handle mutually recursive visitors in a more natural way (see Section 3.8.2 for details).

The reason why we did not use *V S X* in the functional specification, in the first place, is because our specification setting is too weak to "type" the kinds of those types. In essence, the visitor *V* is a type-constructor parametrized by a strategy *S*, but *S* is also a type constructor that is, in turn, parametrized by a visitor. This kind of mutual dependency between the two type constructors means that their kinds cannot be monomorphic. Yet, our Haskell based specification setting only allows monomorphic kinds. In order to specify our visitor library with this more general form of visitors we need a setting that allows *polymorphic kinds*.

In Appendix I we show a different specification setting, using the Omega language (Sheard, 2005), that supports polymorphic kinds and allows us to give a more precise specification to our visitor component.

6.4 Applications of our Work

Programming with Visitors The obvious application of our work is to use the visitor library to program with visitors. There are plenty of applications where visitors can be useful in object-oriented languages. Any problem where adding functions occurs frequently and adding variants is rare can benefit from the use of visitors. The main advantages of using our visitor library instead of just following the traditional design pattern approach are:

1. *Elimination of design choice* - As we have demonstrated in this thesis, the choice of who controls the traversal is parametrizable. If we follow the traditional design pattern approach, we need to choose a particular kind of traversal for our visitor thus loosing flexibility.

2. *Extensibility* - Our visitor library supports extensible visitors and can be used more generally than the original design pattern. In particular, we can use our library on problems where we have interest in adding both new functions and new variants.

3. *Notation* - Our visitor library supports a functional notation for visitors, avoiding the need to use *accept* methods explicitly to express recursion calls. Definitions written in this style look

like definitions by pattern-matching in a functional programming language.

4. *Easy support for DGP* - Our visitor library includes DGP support based on GM. To use of generic functions with some user-defined visitor, we just need to provide a few, boilerplate, definitions. We can even avoid (most of) that boilerplate if we use a sum-of-product visitor instead.

In Scala, the existence of case classes essentially subsumes the need for (traditional) visitors and has the big advantage of having built-in support for definitions using this construct. Although our notation is better than the traditional design pattern, it does not offer the same convenience as case classes. However, case classes force a particular decomposition strategy (basically equivalent to external visitors) and they do not enjoy of the same level of type-safety when programming with extensible datatypes — if we do not provide a catch-all case, it is easy to get run-time errors.

Programming with Recursion Patterns The use of the functional notation with external visitors allow us to have definitions that are, essentially, definitions by pattern matching. For example, if we consider some kind of trees of integers, a possible function that sums all the integers could be written as:

```
def addTree = new VTree[External, int] {
  def empty                          = 0
  def fork (x : int, l : Rec, r : Rec) = x + addTree (l) + addTree (r)
}
```

This style of definition has the advantage of being somewhat intuitive (due to the pattern matching style) but, because recursion is used directly, the recursive pattern in use is not immediately apparent. It is often suggested, as a good programming practise, that recursion patterns should be abstracted and used in definitions instead of this, more direct, style. Recursion patterns are traditionally captured using higher-order functions. For example, we can capture the recursion pattern of *addTree* as a *fold* and define *addTree* in terms of that operation instead.

```
def foldTree[a] (k : a) (f : int ⇒ a ⇒ a ⇒ a) (t : Tree) : a =
  t.accept[External, a] (new VTree[External, a] {
    def empty                          = k
    def fork (x : int, l : Rec, r : Rec) =
      f (x) (foldTree (k) (f) (l)) (foldTree (k) (f) (r))})
def addTree (t : Tree) = foldTree (0) (x ⇒ l ⇒ r ⇒ x + l + r) (t)
```

Although this style is suggested as good practice, there are two problems with it. Firstly, if the programmer has just defined a new datatype he also needs to define the recursion patterns for that datatype. Secondly, many programmers find the usage of higher-order functions like *foldTree* counter-intuitive and are much more comfortable with a direct-style definition by pattern matching. With our library, however, the first problem is solved since recursion patterns are generic and provided automatically with the definition of a new visitor. Furthermore, we also benefit from the library for the second problem because, by using visitors, we get a pattern-matching-like notation even with recursion patterns other than the one given by *External*. For example, using internal visitors (which are equivalent to *folds*) we can write *addTree* in the following way:

$$
\begin{array}{l}
\textbf{def}\ \mathit{addTree} = \textbf{new}\ \mathit{VTree}[\mathit{Internal}, \mathit{int}]\ \{ \\
\quad \textbf{def}\ \mathit{empty} = 0 \\
\quad \textbf{def}\ \mathit{fork}\ (x : \mathit{int}, l : \mathit{Rec}, r : \mathit{Rec}) = x + l + r \\
\}
\end{array}
$$

We believe that, with this programming style, programmers will be more inclined to use recursion patterns because they will not need any additional effort to use them and they will be able to use an intuitive notation like pattern matching.

Less boilerplate and more reuse with DGP There has been a lot of work in functional programming languages on DGP, but very little has been done in object-oriented languages. Still, many uses of generic programming — such as comparing two values for equality, pattern matching on occurrences of a particular string in a value of some datatype, serializing or deserializing values or pretty-printing a value — would have important applications in OO languages. Visitors developed with our library can benefit from generic programming techniques, which leads to increased reuse. Moreover, although the DGP component of our library fits nicely with visitors, it can also be used with other hierarchical structures like case classes, as demonstrated in Section 4.3.3. In essence all that needs to be done is to provide isomorphisms between those structures and the corresponding sum-of-product representation. Furthermore, because classes are essentially records (or named products) these isomorphisms should be easy to define.

Semantics of datatypes Our library allows us to capture a family of visitors and provides the basic notation and tools to define functions over these same visitors. Still, compared to datatypes

found in functional programming languages, the notational overhead required by visitors is relatively large, making them less convenient to use than datatypes. Since visitors are encodings of datatypes, a potential application of our work is to give a semantics for datatypes (and pattern matching) in a possible programming language extension, which would solve the notational problem. Our family of visitors is sufficiently powerful to express (regular) parametric datatypes, mutually recursive datatypes and even datatypes with existential components. This power means that our family of visitors is capable to express most of ML and Haskell 98 algebraic datatypes (the exception being nested datatypes and datatypes with contravariant recursive occurrences in function spaces).

Compiler Optimizations 'The Algebra of Programming' provides a solid ground for reasoning about programs written with recursion patterns. Because each kind of visitor is associated with a recursion pattern, we may hope to use all the algebra around recursion patterns to transform inefficient programs into efficient ones, which is one of the major applications of AoP. Many times, these kind of optimizations can be applied automatically, meaning that a compiler could do it. However, as mentioned before, in the presence of side-effects, those laws do not hold. So in a language like Scala we would need a static analyser that ensured the absence of side-effects before trying to perform any optimizations, but in a language like Haskell this kind of analysis would not be necessary.

6.5 Future Work

In this section we discuss possible future work.

A Purely Functional Object-Oriented Language In Section 6.3 we compared Haskell and Scala and described the advantages (and disadvantages) of each language in the development of visitor and generic programming components. This comparison is helpful to define what would be an ideal language for such a development. As we have argued, although Scala is quite suitable for the development of components, our library could benefit from the absence of side-effects, which is not ensured by Scala. However, this could be easily changed by disallowing side-effects in the first place, making the language pure. A promising starting point would be the work by Cremet *et al.* (2006), which proposes a minimal core calculus that captures many of the Scala programming

language constructs and does not have side-effects. Furthermore, it would also be interesting to add datatypes as an extension, with a semantics inspired by our work on the visitor pattern, and explore the parametrization by decomposition strategy and extensibility on datatypes.

We believe that a *purely functional object-oriented programming language* is an overlooked variation on programming languages — of course, by being purely functional, we could raise the question of whether it remains object-oriented, since, for many people, a crucial point of object-orientation is to have stateful objects. Although there are languages that are purely functional (such as Haskell and Clean) and other languages that are (impure) functional object-oriented (such as OCaml and Scala), there have been little attempts to define languages that are both purely functional and object-oriented — one exception was the prototype language O'Haskell (Nordländer, 2000). Our vision is a language where functions are pure and can be defined with a clean syntax like Haskell; where datatypes can be strategy-parametrizable and extensible; and we have a Scala-like object system (instead of type classes and a Haskell-like module system).

A Library for Design Patterns In the same way that we can have a library for Visitors, we believe that many other design patterns could be expressed as a library using more expressive type systems like the one found in Scala. For example, one other pattern that we have been looking at recently is the Iterator. In Gibbons and Oliveira (2006) we showed that Internal Iterators can be nicely modelled using a kind of effectful traversal based on applicative functors (McBride and Paterson, 2007). Furthermore, because applicative functors have a solid theoretical foundation in terms of so-called *strong lax monoidal functors*, we expect to have a cleaner algebra for reasoning about and optimizing programs using iterators — although again we need to assume the absence of side-effects for the laws to hold. The Iterator pattern can also benefit from generic programming because it is possible to define the effectful traversal function generically (that is, using a generic function). This would be an advantage since, currently, defining a new iterator for some structure is quite ad-hoc.

Therefore, we would like to develop a library supporting the Iterator pattern, as well as exploring the library possibility for other design patterns. Moreover, it would be interesting to explore this library of design patterns in a setting like the one we described before (a purely functional object-oriented language) because we expect that our library components enjoy of many algebraic properties that can only be exploited with the guarantee of absence of side-effects.

Indexed Programming Programming languages are progressively allowing the programmer to express more precise properties of their programs, in a way that compilers can exploit for safety and for efficiency. In particular, developments like *nested datatypes* (Bird and Meertens, 1998), *indexed types* (Xi and Pfenning, 1999), and *GADTs* (Peyton Jones *et al.*, 2006), allow the programmer to specify interesting properties that can be verified at compile-time. For example, we can capture properties about the *shape of data* (such as the dimensions of a matrix, or the balancing of a tree) and the *state of components* (such as safety or security properties of an agent in a protocol) statically. In other words, these language mechanisms allow us to lift properties of programs that would otherwise be available only dynamically, if at all, and make them statically checkable and analysable.

In this thesis we have explored a family of indexed types that we called the (one-parameter) unnested GADTs (or, in other words, GADTs that are not nested datatypes), which has plenty of applications but it is limited in two ways:

- *Nested GADTs* - Our library does not (readily) support nested GADTs, because recursion patterns for nested datatypes are non-trivial to define (Bird and Meertens, 1998; Bird and Paterson, 1999; Hinze, 1999; Martin *et al.*, 2004) and yet our visitor library needs to support different recursion patterns for the datatypes defined with it. However, Johann and Ghani (2007) have recently shown that it is possible to define a generic Church encoding for nested datatypes, which may be good starting point towards removing this limitation of our (indexed) visitor library.

- *Multiple type indexes* - Our library is currently limited to one type index but, ideally, we would like to allow any number of indexes. We know what the general template for multiple arguments is:

$$Composite\ V\ T_1\ \ldots\ T_n \equiv \overbrace{\forall X_{\overline{n}}\ S.\ Decompose\ S \Rightarrow \underbrace{V\ S\ X_{\overline{n}}}_{Visitor} \Rightarrow X_{\overline{n}}\ T_1\ \ldots\ T_n}^{accept\ method}$$

  and we know how to define visitor libraries for particular values of n. However, we have not investigated yet how to translate from that template into a (usable and) linguistically capturable arity-generic visitor library. We believe that there are two possible directions that we can take. The first direction would be a dependently-typed approach where the library

would be parametrized by its arity. The problem that we see with this approach is that, in a language like Scala, parametrization by arity is hard to accomplish and, even if we can do it, the library is probably going to suffer on usability. The second direction would be to create a hierarchy of visitor components where the component handling arity $n + 1$ would refine the component handling arity n. Basically, the type that we need to refine, at each component, is $X_{\overline{n}}$, making use of the fact that $X_{\overline{n+1}} <: X_{\overline{n}} <: X_0$ (here, we use the notation $X_{\overline{n}}$ to indicate the fact that X is a type constructor with n arguments).

If both limitations are overcome, we would basically have a visitor library that is comparable in expressive power to Haskell's GADTs **data** declarations, with the extra advantages of parametrization by decomposition strategy and extensibility. Furthermore, programs that make use of these, so-called, indexed datatypes can be hard to grasp and they often involve additional boilerplate code not present in the non-indexed versions (Gibbons *et al.*, 2007). We believe that DGP and strategy parametrization (itself a form of DGP) may be helpful here because the effort of stating properties can be amortized over more programs.

Theoretical development In Section 6.2 we presented some results of our thesis from a type-theoretic perspective. While we believe that these results are interesting, our presentation is not a proper formal treatment. Such a formalization of our work — for example, using a type-theoretic or categorical treatment — may bring further insights that could be helpful to further simplify and/or increase the expressive power of our library.

Bibliography

Alexandrescu, A. (2001). *Modern C++ Design*. C++ In-Depth Series. Addison-Wesley.

Alimarine, A. and Plasmeijer, M. J. (2001). A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages*, volume 2312 of *Lecture Notes in Computer Science*, pages 168–185. Springer-Verlag.

Aracic, I., Gasiunas, V., Mezini, M., and Ostermann, K. (2006). An overview of CaesarJ. In A. Rashid and M. Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer.

Arnout, K. (2004). *Pattern Componentization*. Ph.D. thesis, Swiss Institute of Technology.

Backhouse, R. C. and Hoogendijk, P. F. (1993). Elements of a relational theory of datatypes. In *Proceedings of the IFIP TC2/WG 2.1 State-of-the-Art Report on Formal Program Development*, Lecture Notes in Computer Science, pages 7–42, Berlin. Springer-Verlag.

Backhouse, R. C., de Bruin, P. J., Hoogendijk, P. F., Malcolm, G., Voermans, E., and van der Woude, J. (1992). Polynomial relators (extended abstract). In *AMAST '91: Proceedings of the Second International Conference on Methodology and Software Technology*, pages 303–326, London, UK. Springer-Verlag.

Bird, R. and Meertens, L. (1998). Nested datatypes. In J. Jeuring, editor, *Proceedings 4th Int. Conf. on Mathematics of Program Construction, MPC'98, Marstrand, Sweden, 15–17 June 1998*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, Berlin.

Bird, R. and Paterson, R. (1999). Generalised folds for nested datatypes. *Formal Aspects of Computing*, **11**(2), 200–222.

Bird, R., de Moor, O., and Hoogendijk, P. (1996). Generic functional programming with types and relations. *Journal of Functional Programming*, **6**(1), 1–28.

Bird, R. S. and De Moor, O. (1997). *Algebra of Programming*, volume 100 of *International Series in Computing Science*. Prentice Hall.

Böhm, C. and Berarducci, A. (1985). Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, **39**(2-3), 135–153.

Bringert, B. and Ranta, A. (2006). A pattern for almost compositional functions. *SIGPLAN Not.*, **41**(9), 216–226.

Brockschmidt, K. (1995). *Inside OLE (2nd ed.)*. Microsoft Press, Redmond, WA, USA.

Broy, M., Deimel, A., Henn, J., Koskimies, K., Plasil, F., Pomberger, G., Pree, W., Stal, M., and Szyperski, C. (1998). What characterizes a software component. *Software Concepts & Tools*, **19**(1), 49 – 56.

Buchlovsky, P. and Thielecke, H. (2005). A type-theoretic reconstruction of the Visitor pattern. In *21st Conference on Mathematical Foundations of Programming Semantics (MFPS XXI)*, volume 4 of *Electronic Notes in Theoretical Computer Science (ENTCS)*.

Chakravarty, M. M. T., Keller, G., and Peyton Jones, S. (2005a). Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA. ACM Press.

Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. (2005b). Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Synposium on Principles of Programming Languages*, pages 1–13. ACM Press.

Cheney, J. and Hinze, R. (2002). A lightweight implementation of generics and dynamics. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 90–104, New York, NY, USA. ACM Press.

Cheney, J. and Hinze, R. (2003). First-class phantom types. Technical report, CUCIS TR2003-1901, Cornell University.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, **58**, 345–363.

Cox, B. J. (1990). Planning the software industrial revolution. *IEEE Software magazine*, **7**(6), 25–33.

Cremet, V., Garillot, F., Lenglet, S., and Odersky, M. (2006). A core calculus for Scala type checking. In *Proceddings of Mathmatical Foundations of Computer Science*, Lecture Notes in Computer Science. Springer-Verlag.

Dijkstra, E. W. (1972). The Humble Programmer. *Communications of the ACM*, **15**(10), 859–866.

Ernst, E. (1999). *gbeta - a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Ph.D. thesis, University of Arhus, Denmark.

Ernst, E. (2004). The expression problem, Scandinavian style. In P. Lahire and e. al., editors, *Proceedings of MASPEGHI 2004*, ISRN I3S/RR-2004-15-FR, Oslo, Norway. Laboratoire I3S, Sophia Antipolis.

Forax, R., Duris, E., and Roussel, G. (2005). Reflection-based implementation of Java extensions: the double-dispatch use-case. In *SAC '05: Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1409–1413, New York, NY, USA. ACM Press.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garrigue, J. (2000). Code reuse through polymorphic variants. In *Workshop on Foundations of Software Engineering, Sasaguri, Japan*.

Garrigue, J. (2004). Typing deep pattern-matching in presence of polymorphic variants. In *JSSST Workshop on Programming and Programming Languages, Gamagori, Japan*.

Gibbons, J. (2003). Patterns in datatype-generic programming. In J. Striegnitz, editor, *Declarative Programming in the Context of Object-Oriented Languages, Uppsala*.

Gibbons, J. (2006). Design patterns as higher-order datatype-generic programs. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, pages 1–12, New York, NY, USA. ACM Press.

Gibbons, J. and Oliveira, B. (2006). The essence of the Iterator pattern. In T. Uustalu and C. McBride, editors, *Mathematically-Structured Functional Programming*, volume 4014 of *Lecture Notes in Computer Science*. Springer-Verlag.

Gibbons, J., Wang, M., and Oliveira, B. (2007). Generic and indexed programming. In M. Morazan, editor, *Trends in Functional Programming*.

Grothoff, C. (2003). Walkabout revisited: The Runabout. In *ECOOP 2003 - Object-Oriented Programming*, pages 103–125. Springer-Verlag.

Hall, C. V., Hammond, K., Jones, S. L. P., and Wadler, P. L. (1996). Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, **18**(2), 109–138.

Harper, R. and Lillibridge, M. (1994). A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR.

Hinze, R. (1999). Polytypic functions over nested datatypes. *Discrete Mathematics and Theoretical Computer Science*, **3**(4), 193–214.

Hinze, R. (2000). Polytypic values possess polykinded types. In R. Backhouse and J. N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction, July 3–5, 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag.

Hinze, R. (2003). Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 245–262. Palgrave Macmillan. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

Hinze, R. (2004). Generics for the masses. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 236–243, New York, NY, USA. ACM Press.

Hinze, R. (2006). Generics for the masses. *Journal of Functional Programming*, **16**(4-5), 451–483.

Hinze, R. and Jeuring, J. (2001). Functional pearl: Weaving a web. *Journal of Functional Programming*, (11(6)), 681–689.

Hinze, R. and Löh, A. (2006). "scrap your boilerplate" revolutions. volume 4014 of *Lecture Notes in Computer Science*, pages 180–208.

Hinze, R. and Löh, A. (2007). Generic programming, now! In R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, editors, *Datatype-Generic Programming, Advanced Lectures*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag.

Hinze, R. and Löh, A. (2008). Generic programming in 3d.

Hinze, R. and Peyton Jones, S. (2000). Derivable type classes. In *Haskell Workshop*.

Hinze, R., Löh, A., and d. S. Oliveira, B. C. (2006). 'Scrap your Boilerplate' reloaded. In *Functional and Logic Programming*.

Holdermans, S., Jeuring, J., Löh, A., and Rodriguez, A. (2006). Generic views on data types. In T. Uustalu, editor, *Proceedings 8th International Conference on Mathematics of Program Construction, MPC 2006*, volume 4014 of *Lecture Notes in Computer Science*, pages 209–234. Springer-Verlag, Berlin.

Howard, W. A. (1980). The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press. Reprint of 1969 article.

Huet, G. (1997). The Zipper. *Journal of Functional Programming*, **7**(5), 549–554.

Hughes, J. (1999). Restricted data types in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28.

Jan Martin Jansen, Pieter Koopman, R. P. (2005). Data types and pattern matching by function application. *Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05*.

Jansson, P. (2000). *Functional Polytypic Programming*. Ph.D. thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden.

Jeuring, J., Yakushev, A. R., Kiselyov, O., Gerdes, A., d. S. Oliveira, B. C., and Jansson, P. (2007). Comparing libraries for generic programming in Haskell. In preparation.

Johann, P. and Ghani, N. (2007). Initial algebra semantics is enough! In S. R. D. Rocca, editor, *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, pages 207–222. Springer.

Jones, S. P., editor (2003). *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England.

Kühne, T. (1999). *A Functional Pattern System for Object-Oriented Design*. Verlag Dr. Kovač, ISBN 3-86064-770-9, Hamburg, Germany.

Lämmel, R. and Peyton Jones, S. (2003). Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, **38**(3), 26–37. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

Lämmel, R. and Peyton Jones, S. (2004). Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2004)*, pages 244–255. ACM Press.

Lämmel, R. and Peyton Jones, S. (2005). Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2005)*, pages 204–215. ACM Press.

Läufer, K. (2003). What functional programmers can learn from the Visitor pattern. Technical report, Loyola University Chicago.

Leroy, X. (1994). Manifest types, modules, and separate compilation. In *Proceedings 21st Annual ACM Symposium on Principles of Programming Languages*, pages 109–122.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. (2005). *The Objective Caml system*.

Löh, A. (2004). *Exploring Generic Haskell*. Ph.D. thesis, Utrecht University.

Löh, A. and Hinze, R. (2006). Open data types and open functions. In *PPDP '06: Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 133–144, New York, NY, USA. ACM Press.

Löh, A., Jeuring, J., and al. (2005). The Generic Haskell users's guide. Technical Report UU-CS-2005-004, Utrecht University.

Malcolm, G. (1990). Data structures and program transformation. *Sci. Comput. Program.*, **14**(2-3), 255–279.

Martin, C., Gibbons, J., and Bayley, I. (2004). Disciplined, efficient, generalised folds for nested datatypes. *Formal Aspects of Computing*, **16**(1), 19–35.

McBride, C. and Paterson, R. (2007). Applicative programming with effects. *Journal of Functional Programming*.

McIlroy, M. (1969). Mass Produced Software Components. In *Proceedings of NATO Conference on Software Engineering*, pages 88–98, New York. Petrocelli/Charter.

McIlroy, M. D., Pinson, E. N., and Tague, B. A. (1978). Unix time-sharing system forward. In *The Bell System Technical Journal. Bell Laboratories.*, page 1902.

Meertens, L. (1992). Paramorphisms. *Formal Aspects of Computing*, **4**(5), 413–425.

Meijer, E., Fokkinga, M., and Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, Berlin.

Meyer, B. (1997). *Object-Oriented Software Construction*. Upper Saddle River, N.J., Prentice Hall, 2nd edition.

Meyer, B. and Arnout, K. (2006). Componentization: The Visitor example. *Computer*, **39**(7), 23–30.

Moors, A., Piessens, F., and Joosen, W. (2006). An object-oriented approach to datatype-generic programming. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN Workshop on Generic Programming*, pages 96–106, New York, NY, USA. ACM Press.

Moors, A., Piessens, F., and Odersky, M. (2007). Towards equal rights for higher-kinded types. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages*.

Naur, P. and Randell, B., editors (1969). *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*.

Nordländer, J. (2000). Polymorphic subtyping in O'Haskell. In *APPSEM Workshop on Subtyping and Dependent Types in Programming*.

Norell, U. and Jansson, P. (2004). Polytypic programming in haskell. In *Implementation of Functional Languages*, volume 3145 of *Lecture Notes in Computer Science*.

Norvig, P. (1996). Design patterns in dynamic programming. In *Object World 96*, Boston, MA.

Nystrom, N., Chong, S., and Myers, A. C. (2004). Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 99–115, New York, NY, USA. ACM Press.

Odersky, M. (2006a). An Overview of the Scala programming language (second edition). Technical Report IC/2006/001, EPFL Lausanne, Switzerland.

Odersky, M. (2006b). Poor man's type classes. `http://lamp.epfl.ch/~odersky/talks/wg2.8-boston06.pdf`.

Odersky, M. (2007a). Scala by example. `http://scala.epfl.ch/docu/files/ScalaIntro.pdf`.

Odersky, M. (2007b). The Scala language specification version 2.4. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`.

Odersky, M. and Zenger, M. (2005a). Independently extensible solutions to the expression problem. In *Proceedings of Foundations of Object-Oriented Languages 12*. `http://homepages.inf.ed.ac.uk/wadler/fool`.

Odersky, M. and Zenger, M. (2005b). Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 41–57, New York, NY, USA. ACM Press.

Oliveira, B. and Gibbons, J. (2005). TypeCase: a design pattern for type-indexed functions. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 98–109, New York, NY, USA. ACM Press.

Oliveira, B. C., Hinze, R., and Löh, A. (2006). Extensible and modular generics for the masses. In *Seventh Symposium on Trends in Functional Programming*, pages 109–138.

Palsberg, J. and Jay, C. B. (1998). The essence of the Visitor pattern. In *Proc. 22nd IEEE Int. Computer Software and Applications Conf., COMPSAC*, pages 9–15.

Parigot, M. (1992). Recursive programming with proofs. *Theoretical Computer Science*, **94**(2), 335–356.

Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. (2006). Simple unification-based type inference for gadts. *SIGPLAN Not.*, **41**(9), 50–61.

Pfister, C. and Szyperski, C. (1996). Why objects are not enough. In *Proceedings, International Component Users Conference*, Munich, Germany. SIGS.

Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2003). Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274. Springer Verlag.

Schinz, M. (2007). A Scala tutorial for Java programmers. `http://scala.epfl.ch/docu/files/ScalaTutorial.pdf`.

Sheard, T. (2005). Putting Curry-Howard to work. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 74–85, New York, NY, USA. ACM Press.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 38–45, New York, NY, USA. ACM Press.

Strachey, C. (1967). Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

Swierstra, W. (2008). Data types à la carte. Accepted for publication in the Journal of Functional Programming.

Szyperski, C. (1996). Independently extensible systems – software engineering potential and challenges. In *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia.

Szyperski, C. (2002). Universe of composition.

Torgersen, M. (2004). The expression problem revisited: Four new solutions using generics. In M. Odersky, editor, *ECOOP 2004—Object-Oriented Programming, 18th European Conference, Oslo, Norway, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143.

Udell, J. (1994). Componentware. In *BYTE Magazine*, pages 46 – 56.

VanDrunen, T. and Palsberg, J. (2004). Visitor-oriented programming. In *Proceedings of FOOL-11, the 11th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New York, NY, USA. ACM Press.

Wadler, P. (1989). Theorems for free! In *Proceedings 4th International Conference on Functional Programming Languages and Computer Architecture, FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York.

Wadler, P. (1993). Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the 1992 Marktoberdorf International Summer School*. Springer-Verlag.

Wadler, P. (1998). The expression problem. Java Genericity Mailing list.

Wadler, P. (2003). A prettier printer. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, pages 223–244. Palgrave Macmillan. ISBN 1-4039-0772-2 hardback, ISBN 0-333-99285-7 paperback.

Weck, W. and Szyperski, C. (1996). Do we need inheritance.

Weirich, S. (2006). RepLib: a library for derivable type classes. In *Haskell '06: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*, pages 1–12, New York, NY, USA. ACM Press.

Xi, H. and Pfenning, F. (1999). Dependent types in practical programming (extended abstract). In A. Aiken, editor, *Proceedings of the Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas.

Xi, H., Chen, C., and Chen, G. (2003). Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 224–235. ACM Press.

Zenger, M. and Odersky, M. (2001). Extensible algebraic datatypes with defaults. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 241–252, New York, NY, USA. ACM Press.

Index

Appendix A

Functional Specification of the VISITOR Library in Haskell

The type μ is expressed in Haskell as:

```
newtype Mu v = Mu (∀a s. Decompose s ⇒ v (s v a) a → a)
```

In Haskell we can use type classes to implement *Decompose s* and make that argument implicit.

The class *Decompose* and the operation *dec* are defined as:

```
class Decompose s where
  dec :: v (s v a) a → Mu v → s v a
```

Internal and *External* visitors and the corresponding instances of *Decompose* are declared as:

```
newtype Internal (v :: ∗ → ∗ → ∗) a = F{get :: a}
newtype External v a                = C (Mu v)
instance Decompose Internal where
  dec t (Mu u) = F (u t)
instance Decompose External where
  dec _        = C
```

We show how to encode natural numbers using the library next

```
data NatF r a = NatF{z :: a, s :: r → a}
type Nat      = Mu NatF
zero  :: Nat
zero  = Mu z
suc   :: Nat → Nat
suc n = Mu (λv → s v (dec v n))
```

and give two examples: the first one, using a Church encoding (i.e. an internal visitor), converts a natural to a built-in integer; the second one using a Parigot encoding (i.e. an external visitor) defines the predecessor function, which is hard to define with a Church encoding.

```
toInt :: Nat → Int
toInt (Mu n) = n visitor
  where visitor :: NatF (Internal NatF Int) Int
        visitor = NatF{z = 0, s = λ(F x) → x + 1}
predessessor :: Nat → Nat
predessessor (Mu n) = n visitor
  where visitor :: NatF (External NatF Nat) Nat
        visitor = NatF{z = error "Ops!", s = λ(C x) → x}
```

Appendix B

Translation of Datatypes

In this appendix we sketch, more formally, how the translation between a datatype-like declaration and visitors proceeds. In Section 3.8 we have examples of the translation scheme. Before we start detailing the translation, we note that the extensively used overline notation $\overline{E}$ is just syntactic sugar to avoid the proliferation of indexes. So, generally we have that:

$$\overline{E} \equiv E_1 \ \ldots \ E_n$$

We start by considering a mutually-recursive set of datatype declarations of the form:

$$\overline{\textbf{data}\ T\ \overline{a} = \overline{\exists\, \overline{e}.\ C\ \overline{t}}}$$

As explained, the overline notation ranging over the **data** declaration means that there are $T_1 \ \ldots \ T_n$ mutually-recursive datatypes. Each individual datatype has a set of type parameters $\overline{a}$ and a set of constructors $\overline{C}$. Each constructor C_i can have a number of existentially-quantified type arguments $\overline{e}$ and has, itself, a set of parameters $\overline{t}$.

We should note that when we mention that we have a set of mutually-recursive set of datatypes, we mean it strictly. For example

data *List a* = *Nil* | *Cons a* (*List a*)
data *Tree a* = *Empty* | *Fork a* (*Tree a*) (*Tree a*)

are not mutually-recursive and the translation of the two datatypes proceeds by considering two sets of declarations independently: the first set with {**data** *List a*} and the second set with {**data** *Tree a*}. In contrast, the following two datatype declarations

data *RoseTree a* = *Fork a* (*Forest a*)

data *Forest a = Nil | Cons* (*RoseTree a*) (*Forest a*)

are mutually-recursive because *RoseTree* depends on *Forest* and vice-versa. So, in this case, the translation proceeds by considering the set {**data** *RoseTree a*, **data** *Forest a*}.

The translation imposes some restrictions on the form of datatype declarations.

- The first restriction is that mutually-recursive datatypes $T_1 \ \ldots \ T_n$ must all have the same number of type arguments $\overline{a}$.
- The second restriction is that we cannot have nested datatypes. For example

  data *PTree a = PEmpty | PFork a* (*PTree* (a, a))

  is not allowed because the recursive ocurrence type *PTree* is nested. This restriction is imposed by the fact that we consider generic encodings. If we considered *Parigot* encodings only, this would not be problematic.
- The third restriction is that we cannot have recursive occurrences on negative positions of functional arguments. For example:

  data *Value = Number Int | Func* (*Value* $\rightarrow$ *Value*)

  is not allowed because a recursive occurrence of *Value* appears at a contravariant position in a function space. Again, the reason for this restriction has to do with the fact that we are considering generic encodings. This would not be a problem if we considered *Parigot* encodings only.
- Finally, the translation does not consider the existence of *datatypes* with higher-ranked types. For example:

  data *GTree g a = Fork a* (*g* (*GTree g a*))

  is not allowed because the type parameter *g* has a higher-ranked type.

For the set of datatypes $T_1 \ \ldots \ T_n$ we generate a corresponding number of concrete visitors (that is, traits that extends *Visitor*) and composites (which are just type synonyms to $Composite[TVisitor_i[\overline{a}]]$). This step of the translation is shown next:

$$\overline{\textbf{trait } \mathit{TVisitor}[\overline{a}] \textbf{ extends } \mathit{Visitor}\ \{}$$
$$\quad \overline{\textbf{def } \mathit{mref}_{\mathbf{D}}\ : \mathit{TVisitor}_{\mathbf{D}}[\overline{a}]\ \{\textbf{type } X = \mathit{TVisitor}.\textbf{this}.X; \textbf{type } S = \mathit{TVisitor}.\textbf{this}.S\}}$$
$$\quad \overline{\textbf{def } c[\overline{e}]\ \textbf{genType}\ (\ \overline{\textbf{data } T\ \overline{a}}\ ,\ \overline{t}_{\mathbf{C}}) : X}$$
$$\}$$
$$\overline{\textbf{type } T[\overline{a}] = \mathit{Composite}[\mathit{TVisitor}[\overline{a}]]}$$

Each of the visitors $\mathit{TVisitor}_i[\overline{a}]]$ has $n-1$ mref_D , which are the mutually-recursive references to other visitors. For example, the types *List* and *Tree* above would have 0 *mref* methods because they do not have any mutually-recursive references. However, in the case of *RoseTree* and *Forest*, there would be one reference to the *RoseTreeVisitor* visitor in *ForestVisitor* and one reference to the *ForestVisitor* in *RoseTreeVisitor*. For each constructor C_i of the datatype we would also have a corresponding method c_i with its parameter body being computed by **genType**, which we discuss later.

For each constructor C_j on a datatype T_i we need to generate a constructor definition. We outline this translation next:

$$\overline{\overline{\textbf{def } C[\overline{e}, \overline{a}]\ (\overline{t}_{\mathbf{C}}) : T[\overline{a}] = \textbf{new } T[\overline{a}]\ \{}$$
$$\quad \textbf{def } \mathit{accept}[s <: \mathit{Strategy}, x]\ (\mathit{vis} : \mathit{TVisitor}[\overline{a}]\ \{\textbf{type } X = x; \textbf{type } S = s\})$$
$$\qquad\qquad (\textbf{implicit } \mathit{decompose} : \mathit{Decompose}[s]) : x =$$
$$\qquad \mathit{vis}.c[\overline{e}]\ \textbf{genBody}\ (\mathit{vis},\ \overline{\textbf{data } T\ \overline{a}}\ ,\ \overline{t}_C)$$

A constructor C_j has a set of type arguments composed of the sets $\overline{e}$ and $\overline{a}$ and its type signature is just a syntactic variation $\overline{t}_C$ of $\overline{t}$ properly adapted to Scala's syntax. The *accept* method of *Composite* is implemented by invoking the corresponding method c_j on the visitor and the arguments of c_j are computed by **genBody** that we discuss next.

The type and the bodies of the visit methods c_j corresponding to a constructor C_j are, respectively, computed by the functions **genType** and **genBody** defined as follows:

$$\textbf{genType}\ (\ \overline{\textbf{data } T\ \overline{a}}\ ,\ \overline{t}\ ) =$$
$$\quad \{v : \textbf{case } \mathit{statusOf}\ (t')\ \textbf{of}$$
$$\qquad\qquad \mathit{Recursive} \ \rightarrow R[\mathit{TVisitor}[\overline{a}]]$$
$$\qquad\qquad \mathit{MutualRec} \rightarrow R[\mathit{TVisitor}_D[\overline{a}]]$$
$$\qquad\qquad \mathit{NonRec} \ \rightarrow t'$$
$$\qquad\qquad \mid t' \leftarrow \overline{t}, v \leftarrow \mathit{fresh}\}$$
$$\textbf{genBody}\ (\mathit{vis},\ \overline{\textbf{data } T\ \overline{a}}\ ,\ \overline{t}\ ) =$$
$$\quad \{\textbf{case } \mathit{statusOf}\ (t')\ \textbf{of}$$
$$\qquad \mathit{Recursive} \ \rightarrow \mathit{decompose.dec}[\mathit{TVisitor}[\overline{a}], x]\ (\mathit{vis}, v)$$
$$\qquad \mathit{MutualRec} \rightarrow \mathit{decompose.dec}[\mathit{TVisitor}_D[\overline{a}], x]\ (\mathit{vis}.\mathit{mref}_D, v)$$

$$\begin{array}{l} \mathit{NonRec} \quad \rightarrow v \\ \mid t' \leftarrow \overline{t}, v \leftarrow \mathit{fresh}\} \end{array}$$

Both functions take the set of mutually-recursive datatypes and the set of constructor arguments $\overline{t}$ into account; and both functions are defined by checking whether each type t' in $\overline{t}$ is either *recursive* (a reference to the same datatype that is being defined); or a mutually-recursive reference (a reference to one of the datatypes in the set of mutually-recursive definitions); or if it it just non-recursive.

Appendix C

Paramorphic Visitors Specification

In this appendix, we present the functional specification for paramorphic visitors. We start by defining a datatype for the decomposition strategy and the corresponding instance of *Decompose*:

```
newtype Para v a = P (a, Mu v)
instance Decompose Para where
  dec t (Mu u) = P (u t, Mu u)
```

We then define the functions *plus* and *times* that will be used to define factorial.

```
plus :: Nat → Nat → Nat
plus (Mu n) m = n visitor
  where visitor :: NatF (Internal NatF Nat) Nat
        visitor = NatF{z = m, s = λ(F x) → succ x}
times :: Nat → Nat → Nat
times (Mu n) m = n visitor
  where visitor :: NatF (Internal NatF Nat) Nat
        visitor = NatF{z = zero, s = λ(F x) → plus x m}
```

Finally, we define factorial as:

```
fact :: Nat → Nat
fact (Mu n) = n visitor
  where visitor :: NatF (Para NatF Nat) Nat
        visitor = NatF{z = succ zero, s = λ(P (x, y)) → times x (succ y)}
```

Appendix D

Paramorphic Visitors

Here we present the full Scala code for Section 3.8.4. The paramorphic decomposition strategy is defined with the following code:

```
trait Para extends Strategy {
    type Y = Pair[X, Composite[V]]
}
implicit def para : Decompose[Para] = new Decompose[Para] {
  def dec[v <: Visitor, x] (vis : v {type X = x; type S = Para}, comp : Composite[v]) =
    new Para {type V = v; type X = x;
                def get = Pair[x, Composite[v]] (comp.accept (vis), comp)}
}
implicit def para2pair[v <: Visitor, x]
  (x : Para {type V >: v; type X = x}) : Pair[x, Composite[v]] = x.get
```

A visitor for peano numerals, that is going to be used by our example, can be defined as follows:

```
trait NatVisitor extends Visitor {
  type Rec = R[NatVisitor]
  def zero : X
  def succ (n : Rec) : X
    }
    type Nat = Composite[NatVisitor]
case class Zero extends Nat {
  def accept[s <: Strategy, x] (vis : NatVisitor {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
    vis.zero
}
case class Succ (n : Nat) extends Nat {
  def accept[s <: Strategy, x] (vis : NatVisitor {type X = x; type S = s})
                               (implicit decompose : Decompose[s]) : x =
```

```
      vis.succ (decompose.dec (vis, n))
  }
  abstract class VNat[s <: Strategy, b] (implicit decompose : Decompose[s])
    extends VisitorFunc[NatVisitor, s, b] (decompose)
    with NatVisitor
```

Next we implement addition and multiplication on naturals; and *fst* and *snd* on pairs.

```
  def plus (m : Nat) : Nat ⇒ Nat = new VNat[Internal, Nat] {
    def zero          = m
    def succ (n : Rec) = Succ (n)
  }
  def mult (m : Nat) = new VNat[Internal, Nat] {
    def zero          = Zero
    def succ (n : Rec) = plus (n) (m)
  }
  def fst[a, b] (x : Pair[a, b]) : a = x match {case Pair (f, s) ⇒ f }
  def snd[a, b] (x : Pair[a, b]) : b = x match {case Pair (f, s) ⇒ s}
```

Finally, we can define the factorial function as:

```
  def fact : Nat ⇒ Nat = new VNat[Para, Nat] {
    def zero          = Succ (Zero)
    def succ (n : Rec) = mult (fst (n)) (Succ (snd (n)))
  }
```

Appendix E

Serialization Library

This is the full code for the serialization library presented in Section 4.6.

```
object Serialization {
  def repeat (c : char, times : int) : String = {
    var sb : StringBuffer = new StringBuffer ()
    var t = times
    while (t > 0) {
      sb.append (c);
      t = t − 1;
    }
    return sb.toString ();
  }
  def encodeIntegral (x : int, size : int) : String = {
    def s = Integer.toBinaryString (x);
    return (repeat ('0', size − s.length ()) + s)
  }
  def encodeInt (x : int) = encodeIntegral (x, 32)
  def encodeChar (x : char) = encodeIntegral (x.asInstanceOf [int], 8)
  def exp (x : int, y : int) : int = {
    var i = 0;
    var value = 1;
    while (i < y) {
      value = x ∗ value;
      i = i + 1;
    }
    return value;
  }
  def decodeIntegral (x : String) : int = {
    def sb : StringBuffer = new StringBuffer (x).reverse ();
    def len = x.length ();
```

```
    var value = 0;
    var i : int = 0;
    while (i < len) {
      if (sb.charAt (i).equals ('1')) value = exp (2, i) + value;
      i = i + 1;
    }
    return value;
  }
  def decodeInt (x : String) : Prod[int, String] = {
    def rest : String = x.substring (32)
    def value = x.substring (0, 32)
    return Prod (decodeIntegral (value), rest)
  }
  def decodeChar (x : String) : Prod[char, String] = {
    def rest : String = x.substring (8)
    def value = x.substring (0, 8)
    return Prod (decodeIntegral (value).asInstanceOf [char], rest)
  }
    // A new generic function
    trait Serialize extends TypeConstructor {
      def serialize (x : A) : String
    }
    trait MySerialize extends Generic {
      type G = Serialize
      def unit = new Serialize {type A = One; def serialize (x : A) = ""}
      def int = new Serialize {type A = int; def serialize (x : A) = encodeInt (x)}
      def char = new Serialize {type A = char; def serialize (x : A) = encodeChar (x)}
      def plus[a, b] (a : G {type A = a}, b : G {type A = b}) = new Serialize {
        type A = Plus[a, b]
        def serialize (x : A) = x.accept (new PlusVisitor[a, b, String] {
          def inl (y : a) = "0" + a.serialize (y)
          def inr (z : b) = "1" + b.serialize (z)
        })
      }
      def prod[a, b] (a : G {type A = a}, b : G {type A = b}) = new Serialize {
        type A = Prod[a, b]
        def serialize (x : A) = x.accept (new ProdVisitor[a, b, String] {
          def prod (y : a, z : b) = a.serialize (y) + b.serialize (z)
        })
      }
      def view[a, b] (iso : Iso[b, a], a: ⇒ G {type A = a}) = new Serialize {
        type A = b
        def serialize (x : A) = a.serialize (iso.from (x))
      }
    }
```

```
def testVal = Cons (3, Cons (4, Nil[int]))
implicit object mySerial extends MySerialize
def serial[t] (x : t) (implicit r : Rep[t]) : String =
  r.rep (mySerial).serialize (x)
def serialSumProd[f <: TypeConstructor] (x : SumProd[f]) (implicit fr : FRep[f]) =
    x.accept[Internal, String] (new VSumProd[Internal, f, String] {
      def seriala = new Serialize () {
        type A = Rec
        def serialize (x : A) : String = x
      }
      def visit (x : f {type A = Rec}) = fr.frep[Serialize, Rec] (seriala).serialize (x)
    })
def testSer = serialSumProd (testVal) (listFRep)
  trait DeSerialize extends TypeConstructor {
    def deSerialize (x : String) : Prod[A, String]
  }
  trait MyDeSerialize extends Generic {
    type G = DeSerialize
    def unit = new DeSerialize {
      type A = One;
      def deSerialize (x : String) = Prod[A, String] (One, "")
    }
    def int = new DeSerialize {
      type A = int;
      def deSerialize (x : String) = decodeInt (x)
    }
    def char = new DeSerialize {
      type A = char;
      def deSerialize (x : String) = decodeChar (x)
    }
    def plus[a, b] (a : G {type A = a}, b : G {type A = b}) = new DeSerialize {
      type A = Plus[a, b]
      def deSerialize (x : String) : Prod[Plus[a, b], String] =
        if (x.substring (0, 1).equals ("0")) {
          def prod = a.deSerialize (x.substring (1))
          return Prod (Inl (prod.fst), prod.snd)
        } else {
          def prod = b.deSerialize (x.substring (1))
          return Prod (Inr (prod.fst), prod.snd)
        }
    }
    def prod[a, b] (a : G {type A = a}, b : G {type A = b}) = new DeSerialize {
      type A = Prod[a, b]
      def deSerialize (x : String) : Prod[Prod[a, b], String] = {
        def prod1 = a.deSerialize (x)
```

```
            def prod2 = b.deSerialize (prod1.snd)
            return Prod (Prod (prod1.fst, prod2.fst), prod2.snd)
          }
        }
        def view[a, b] (iso : Iso[b, a], a: ⇒ G {type A = a}) = new DeSerialize {
          type A = b
          def deSerialize (x : String) : Prod[b, String] = {
            def prod = a.deSerialize (x)
            return Prod (iso.to (prod.fst), prod.snd)
          }
        }
      }
      implicit object myDeSerial extends MyDeSerialize
      def deSerial[t] (x : String) (implicit r : Rep[t]) : t =
        r.rep (myDeSerial).deSerialize (x).fst
      def deserialAux = new DeSerialize {
        type A = String
        def deSerialize (x : String) : Prod[A, String] = Prod (x, "")
      }
      def deSerialSumProd[f <: TypeConstructor]
        (x : String) (implicit fr : FRep[f], funct : Functor[f]) : SumProd[f] =
          ana[f, String] ((y : String) ⇒
            fr.frep[DeSerialize, String] (deserialAux).deSerialize (y).fst) (x)
}
```

Appendix F

Functional Specification for Indexed Visitors

Here we present the Haskell code for the functional specification of indexed visitors, starting with the composite that captures our family of indexed visitors is given by:

```
newtype Mu v t = Mu (∀a s. Decompose s ⇒ v (s v a) a → a t)
```

As before, we have two strategies (one for internal and one for external visitors).

```
newtype Internal (f :: (∗ → ∗) → (∗ → ∗) → ∗) a t = F{get :: a t}
newtype External f (a :: ∗ → ∗) t = C{getC :: Mu f t}
```

The class *Decompose* and its two instance are defined as:

```
class Decompose s where
  dec :: v (s v a) a → Mu v t → s v a t
instance Decompose Internal where
  dec t (Mu u) = F (u t)
instance Decompose External where
  dec _ = C
```

Now we show one concrete example of an indexed visitor that can be captured with this new version of the library:

```
data Generic r g = Generic{
  unit  :: g (),
  int   :: g Int,
  prod :: ∀a b. r a → r b → g (a, b)
}
```

The *Generic* record is a simplified version of the visitor used in GM. We can also encode *Rep* as follows:

```
class Rep t where
  rep :: Decompose s ⇒ Generic (s Generic g) g → g t
instance Rep () where
  rep = unit
instance Rep Int where
  rep = int
instance (Rep a, Rep b) ⇒ Rep (a, b) where
  rep gen = prod gen (dec gen (Mu rep)) (dec gen (Mu rep))
```

Finally we show how to define a generic function that adds all the integers contained in a structure using internal visitors and external visitors.

```
newtype GCount t = GCount{count :: t → Int}
gcount :: Rep t ⇒ t → Int
gcount = count (rep $ Generic{
      unit = GCount (const 0),
      int = GCount id,
      prod = λ(F ra) (F rb) → GCount (λ(x, y) → count ra x + count rb y)})
gcount2 :: Rep t ⇒ t → Int
gcount2 = count (rep func)
  where func = Generic{
      unit = GCount (const 0),
      int = GCount id,
      prod = λ(C (Mu ra)) (C (Mu rb)) →
        GCount (λ(x, y) → count (ra func) x + count (rb func) y)}
```

Appendix G

Functional Specification for the Family of Sums of Products

The following code defines a visitor that can handle sums of products. We first define, as usual, a concrete visitor *SumProdF* and a composite *SumProd*.

```
newtype SumProdF f r a = SumProd{visit :: f r → a}
type SumProd f         = Mu (SumProdF f)
```

The constructor *inF* and the deconstructor *outF* are given by:

```
inF :: Functor f ⇒ f (SumProd f) → SumProd f
inF t = Mu (λv → visit v (fmap (dec v) t))
outF :: Functor f ⇒ SumProd f → f (SumProd f)
outF = get ∘ dec (SumProd (fmap (inF ∘ get)))
```

We consider parametric lists as one example of a datatype that can be defined using our visitor based on sums of products.

```
newtype ListF a r = ListF{listF :: Either () (a, r)}
type List a = SumProd (ListF a)
instance Functor (ListF a) where
  fmap f (ListF (Left ()))       = ListF (Left ())
  fmap f (ListF (Right (x, xs))) = ListF (Right (x, f xs))
nil       :: List a
nil       = inF (ListF (Left ()))
cons      :: a → List a → List a
cons x xs = inF (ListF (Right (x, xs)))
```

Finally we show how we can define recursion patterns:

```
cata :: Functor f ⇒ (f a → a) → SumProd f → a
cata v (Mu m) = m (SumProd (v ∘ fmap get))
ana :: Functor f ⇒ (a → f a) → a → SumProd f
ana c x = Mu (λv → visit v (fmap (dec v ∘ ana c) (c x)))
```

Appendix H

Extensible Visitors Using Abstract Types

Here we present the code for Section 5.5. We start with the trait *ExtensibleGMVisitor*:

```
trait ExtensibleGMVisitor {
  type Gen <: Generic
  type GenWith[x <: TypeConstructor, s <: Strategy] = Gen {type X = x; type S = s}
  trait Generic extends Visitor {
    type Rec[t] = R[Gen, t]
    def unit  : X {type A = One}
    def int   : X {type A = int}
    def char : X {type A = char}
    def plus[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Plus[a, b]}
    def prod[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Prod[a, b]}
    def view[a, b] (iso : Iso[b, a], a: ⇒ Rec[a]) : X {type A = b}
  }
  type Rep[T] = Composite[Gen, T]
  implicit def RUnit = new Rep[One] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
        : x {type A = One} = vis.unit
  }
  implicit def RInt = new Rep[int] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
        : x {type A = int} = vis.int
  }
  implicit def RChar = new Rep[char] {
    def accept[s <: Strategy, x <: TypeConstructor]
      (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
        : x {type A = char} = vis.char
  }
```

```
    implicit def RPlus[a, b] (implicit a : Rep[a], b : Rep[b]) : Rep[Plus[a, b]] =
      new Rep[Plus[a, b]] {
        def accept[s <: Strategy, x <: TypeConstructor]
          (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
            : x {type A = Plus[a, b]} =
              vis.plus (decompose.dec (vis, a), decompose.dec (vis, b))
      }
    implicit def RProd[a, b] (implicit a : Rep[a], b : Rep[b]) = new Rep[Prod[a, b]] {
      def accept[s <: Strategy, x <: TypeConstructor]
        (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
          : x {type A = Prod[a, b]} =
            vis.prod (decompose.dec (vis, a), decompose.dec (vis, b))
      }
    trait Size extends TypeConstructor {
      def size (x : A) : int
      }
    trait MySize requires GenWith[Size, Internal] extends Generic {
      type X = Size
      type S = Internal
      def unit = new Size {type A = One; def size (x : A) = 0}
      def int = new Size {type A = int; def size (x : A) = 0}
      def char = new Size {type A = char; def size (x : A) = 0}
      def plus[a, b] (a : Rec[a], b : Rec[b]) = new Size {
        type A = Plus[a, b]
        def size (x : A) = x.accept (new PlusVisitor[a, b, int] {
          def inl (y : a) = a.get.size (y)
          def inr (z : b) = b.get.size (z)
        })
      }
      def prod[a, b] (a : Rec[a], b : Rec[b]) = new Size {
        type A = Prod[a, b]
        def size (x : A) = x.accept (new ProdVisitor[a, b, int] {
          def prod (y : a, z : b) = a.get.size (y) + b.get.size (z)
        })
      }
      def view[a, b] (iso : Iso[b, a], a: ⇒ Rec[a]) = new Size {
        type A = b
        def size (x : A) = a.get.size (iso.from (x))
      }
    }
  }
```

Next we present the code required to support lists:

```
  trait ExtensibleGMList extends ExtensibleGMVisitor {
    type Gen <: Generic
```

```
    trait Generic extends super.Generic {
        def list[a] (a : Rec[a]) : X {type A = List[a]}
    }
    implicit def RList[a] (implicit a : Rep[a]) : Rep[List[a]] = new Rep[List[a]] {
      def accept[s <: Strategy, x <: TypeConstructor]
        (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
          : x {type A = List[a]} =
            vis.list[a] (decompose.dec (vis, a))
    }
  }
```

The trait *ExtensibleGMConstr* adds support for meta-information.

```
  trait ExtensibleGMConstr extends ExtensibleGMVisitor {
    type Gen <: Generic
    trait Generic extends super.Generic {
      def constr[a] (name : String, arity : int, g : Rec[a]) : X {type A = a}
    }
    def RConstr[a] (name : String, arity : int, a : Rep[a]) : Rep[a] = new Rep[a] {
      def accept[s <: Strategy, x <: TypeConstructor]
        (vis : Gen {type X = x; type S = s}) (implicit decompose : Decompose[s])
          : x {type A = a} =
            vis.constr (name, arity, decompose.dec (vis, a))
    }
    trait PPrint extends TypeConstructor {
      def pprint (x : A) : Document
    }
    trait GenericPrint requires GenWith[PPrint, External] extends Generic {
      type X = PPrint
      type S = External
      def unit = new PPrint {type A = One; def pprint (x : A) = empty}
      def int = new PPrint {type A = int; def pprint (x : A) = text (x.toString ())}
      def char = new PPrint {type A = char; def pprint (x : A) = text (x.toString ())}
      def plus[a, b] (a : Rec[a], b : Rec[b]) = new PPrint {
          type A = Plus[a, b]
          def pprint (x : A) = x.accept (new PlusVisitor[a, b, Document] {
            def inl (y : a) = a.get.accept (GenericPrint.this).pprint (y)
            def inr (z : b) = b.get.accept (GenericPrint.this).pprint (z)
          })
        }
      def prod[a, b] (a : Rec[a], b : Rec[b]) = new PPrint {
          type A = Prod[a, b]
          def pprint (x : A) = x.accept (new ProdVisitor[a, b, Document] {
            def prod (y : a, z : b) = a.get.accept (GenericPrint.this).pprint (y) :: break ::
                                      b.get.accept (GenericPrint.this).pprint (z)
```

```
            })
          }
        def view[a, b] (iso : Iso[b, a], a: ⇒ Rec[a]) = new PPrint {
          type A = b
          def pprint (x : A) = a.get.accept (GenericPrint.this).pprint (iso.from (x))
        }
        def constr[a] (name : String, arity : int, a : Rec[a]) = new PPrint {
          type A = a
          def s = text (name)
          def pprint (x : A) = if (arity ≡ 0) s else
            group (nest (1, (text ("(") :: s :: break ::
            a.get.accept (GenericPrint.this).pprint (x) :: text (")"))))
        }
      }
      def pretty[t] (x : t) (implicit r : Rep[t], v : GenWith[PPrint, External]) = {
        var writer = new OutputStreamWriter (System.out);
        r.accept (v).pprint (x).format (80, writer);
        writer.flush ();
      }
      def prettyDoc[t] (x : t) (implicit r : Rep[t], v : GenWith[PPrint, External]) =
        r.accept (v).pprint (x)
    }
```

The trait *ExtensibleGMListConstr* uses mixin composition to merge meta-information and list support.

```
    trait ExtensibleGMListConstr extends ExtensibleGMConstr with ExtensibleGMList {
      type Gen <: Generic
      trait Generic extends super[ExtensibleGMConstr].Generic
        with super[ExtensibleGMList].Generic {
      }
      trait GenericPrint requires GenWith[PPrint, External]
        extends super.GenericPrint with Generic {
          override def list[a] (a : Rec[a]) : PPrint {type A = List[a]} = new PPrint {
            type A = List[a]
            def pprint (x : A) = pprintl (x) (a.get, GenericPrint.this)
          }
      }
      def pprintl[a] (x : List[a]) (implicit a : Rep[a], v : GenWith[PPrint, External])
        : Document = x.accept[E, Document] (new VList[E, a, Document] {
          def rest (l : List[a]) : Document =
            l.accept[I, Document] (new VList[I, a, Document] {
              def nil = text ("]")
              def cons (y : a, ys : Rec) = text (", ") :: break :: a.accept (v).pprint (y) :: ys.get
            })
```

```
        def nil = text ("[]")
        def cons (y : a, ys : Rec) = group (nest (1, text ("[") :: a.accept (v).pprint (y) ::
                                                  rest (ys.get)))
    })
}
```

Finally we show how to support the string notation. This code corresponds to the workaround that we had to use since the solution presented in Section 5.5 fails with the following error:

```
name clash between defined and inherited member:
method list:[a](GenericPrint.this.Rec[a])
               thesis.Chapter5.PPrint{type A =
                   thesis.Chapter3.VisitorLib.Composite[
                      thesis.Chapter4.Lists.ListVisitor[a]]} and
method list:[a](GenericPrint.this.Rec[a])
               thesis.Chapter5.PPrint{type A =
                   thesis.Chapter3.VisitorLib.Composite[
                      thesis.Chapter4.Lists.ListVisitor[a]]}
in trait GenericPrint have same type after erasure:
(thesis.Chapter4.IndexedVisitorLib#External)thesis.Chapter5.PPrint
```

when trying to compile the trait *GenericPrint*.

```
trait GenericPrint requires GenWith[PPrint, External] extends super.GenericPrint {
  override def list[a] (a : Rec[a]) : PPrint {type A = List[a]} = new PPrint {
    type A = List[a]
    def pprint (x : A) = pprintl (x) (a.get, GenericPrint.this)
  }
  override def constr[a] (name : String, arity : int, a : Rep[a]) = new PPrint {
    type A = a
    def s = text (name)
    def pprint (x : A) = if (arity ≡ 0) s else
      group (nest (1, (text ("(") :: s :: break ::
      a.rep (GenericPrint.this).pprint (x) :: text (")"))))
  }
}
```

We are unsure of what the problem is here and the error message is not very helpful since the reported signatures for the methods are the same! Nonetheless, the workaround consists of adding string support at the same time we merge meta-information and list support.

```
trait ExtensibleGMStringHack extends ExtensibleGMConstr with ExtensibleGMList {
  type Gen <: Generic
  trait Generic extends super[ExtensibleGMConstr].Generic
    with super[ExtensibleGMList].Generic
  def pprintl[a] (x : List[a]) (implicit a : Rep[a], v : GenWith[PPrint, External])
     : Document = x.accept[E, Document] (new VList[E, a, Document] {
       def rest (l : List[a]) : Document =
         l.accept[I, Document] (new VList[I, a, Document] {
           def nil = text ("]")
           def cons (y : a, ys : Rec) = text (",") :: break :: a.accept (v).pprint (y) :: ys.get
         })
       def nil = text ("[]")
       def cons (y : a, ys : Rec) = group (nest (1, text ("[") :: a.accept (v).pprint (y) ::
                                           rest (ys.get)))
  })
  trait PPrintList extends TypeConstructor {
       def pprintList (x : List[A]) : Document
  }
  implicit def defPrint : GenWith[PPrint, External]
  // A generic function with default
  trait GenericDefault extends Generic {
    type S = External
    def deflt[a] (a : Rep[a]) : X {type A = a}
    def unit  = deflt (RUnit)
    def int   = deflt (RInt)
    def char = deflt (RChar)
    def plus[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Plus[a, b]} =
         deflt (RPlus (a.get, b.get))
    def prod[a, b] (a : Rec[a], b : Rec[b]) : X {type A = Prod[a, b]} =
         deflt (RProd (a.get, b.get))
    def constr[a] (name : String, arity : int, a : Rec[a]) : X {type A = a} =
         deflt (a.get)
    override def list[a] (a : Rec[a]) : X {type A = List[a]} = deflt (RList (a.get))
  }
  trait GenericPrintList requires (GenWith[PPrintList, External] with GenericDefault)
    extends GenericDefault {
      type X = PPrintList
      def deflt[a] (a : Rep[a]) = new PPrintList {
        type A = a;
        def pprintList (x : List[A]) = prettyDoc (x) (RList (a), defPrint)
      }
      override def char = new PPrintList {
        type A = char
        def pprintList (x : List[char]) =
```

```
            text ("\"" + (x.accept[I, String] (new VList[I, Char, String] {
              def nil                  = "\""
              def cons (y : A, ys : Rec) = y.toString () + ys
            })))
          }
          def view[a, b] (iso : Iso[b, a], a: ⇒ Rec[a]) = new PPrintList {
            type A = b
            def pprintList (x : List[A]) =
              pprintl (
                x.accept[I, List[a]] (new VList[I, A, List[a]] {
                  def nil = Nil[a]
                  def cons (x : A, xs : Rec) = Cons (iso.from (x), xs.get)
                })) (a.get, defPrint)
          }
        }
        // object printList
        implicit def defPrintList : GenWith[PPrintList, External]
        def prettyListDoc[t] (x : List[t]) (implicit r : Rep[t], v : GenWith[PPrintList, External]) =
          r.accept (v).pprintList (x)
        trait GenericPrint requires GenWith[PPrint, External]
          extends Generic with super.GenericPrint {
            override def list[a] (a : Rec[a]) : X {type A = List[a]} = new PPrint {
              type A = List[a]
              def pprint (x : A) = prettyListDoc (x) (a.get, defPrintList)
            }
        }
      }
```

Appendix I

A Functional Specification in Omega

The Omega programming language has a syntax inspired by Haskell. The code that we will show in this appendix should, therefore, be fairly familiar to a Haskell programmer, expect that we use GADT-style, instead of traditional Haskell, datatype declarations. Also, due to the stratified kind system that omega has, $*0$ denotes the type of types (the conventional $*$ in Haskell) and $*1$ denotes the type of kinds (more generally, $*(n+1)$ denotes the type of "kinds" at the inferior level $*n$).

The type *Mu*, which models the composite component, is expressed in Omega as:

```
data Mu :: ∀(k :: *1). k ⇝ *0 where
  In :: (∀x s. (∀v a. v s a → Mu v → Wrap s v a) → v s x → x) → Mu v
```

Note that the first argument for *Mu* is kind polymorphic, taking a type constructor of any kind as an argument and we use the type $\forall v\ a.v\ s\ a \rightarrow Mu\ v \rightarrow Wrap\ s\ v\ a$ directly instead of the type-synonym *Decompose s*. Also, due to what it seems a bug in the type checker, the type *Wrap* (which is basically the identity) is needed to convince the type checker that *Mu* is well-typed.

```
data Wrap :: ∀(k1 :: *1) (k2 :: *1). k1 ⇝ k2 ⇝ *0 ⇝ *0 where
  W :: s v a → Wrap s v a
unwrap :: Wrap s v a → s v a
unwrap (W w) = w
```

The *Internal* and *External* strategies are the two corresponding decompositions values are declared as:

```
data Internal :: ∀(k :: *1). k ⇝ *0 ⇝ *0 where
  F :: a → Internal v a
data External :: ∀(k :: *1). k ⇝ *0 ⇝ *0 where
```

```
    C :: Mu v → External v a
external :: v External a → Mu v → Wrap External v a
external v mu = W (C mu)
internal :: v Internal a → Mu v → Wrap Internal v a
internal v (In m) = W (F (m internal v))
```

Again, note the usage of polymorphic kinds to type the first argument (the visitor) of *Internal* and *External*.

We show how to encode natural numbers using the library next.

```
data NatF :: ∀(k :: ∗1). k ⇝ ∗0 ⇝ ∗0 where
    NatF :: a → (s NatF a → a) → NatF s a
type Natural = Mu NatF
zero :: Natural
zero = In (λdec (NatF z s) → z)
succ :: Natural → Natural
succ n = In (λdec (NatF z s) → s (unwrap (dec (NatF z s) n)))
```

and give two examples: the first one, using a Church encoding/visitor, converts a natural to a built-in integer; the second one using a Parigot encoding defines the predecessor function (hard to define with Church encodings).

```
toInt :: Natural → Int
toInt (In n) = n internal visitor
    where visitor :: NatF Internal Int
          visitor = NatF 0 (λ(F x) → x + 1)
pred :: Natural → Natural
pred (In n) = n external visitor
    where visitor :: NatF External Natural
          visitor = NatF (zero) (λ(C x) → x)
```