The VISITOR Pattern as a Reusable, Generic, Type-Safe Component

Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{bruno,menw,jg}@comlab.ox.ac.uk

Abstract

The VISITOR design pattern shows how to separate the structure of an object hierarchy from the behaviour of traversals over that hierarchy. The pattern is very flexible; this very flexibility makes it difficult to capture the pattern formally.

We show how to capture the essence of the VISITOR pattern as a reusable software library, by using advanced type system features appearing in modern object-oriented languages such as Scala. We preserve type-safety statically: no reflection or similar mechanisms are used. The library is generic, in two senses: by types (the traversal return type and the object hierarchy shape) and by strategy (internal versus external control, imperative versus functional behaviour, orthogonal aspects such as tracing and memoisation). Finally, we propose a generalised datatype-like notation, providing a convenient functional decomposition style in object-oriented languages.

1. Introduction

A software component is a piece of software that can be safely reused and flexibly adapted. Safety can be ensured, for example, by a type system; flexibility stems from making components parametrizable. Component-oriented programming [McIlroy, 1969], a programming style in which software is assembled from independent components, has for a long time been advocated as a solution to the so-called software crisis [Naur and Randell, 1969].

This vision still has not been fully realised, largely due to limitations of current programming languages. For example, regarding the structure of datatype definitions, most languages have a bias towards either object-oriented decomposition (where adding new variants is easy) or functional decomposition (where adding new functions is easy). This is an instance of what Tarr et al. [1999] call ‘the tyranny of the dominant decomposition’: when software can be modularized along just one primary dimension at a time, concerns that do not break down naturally along that dimension will be scattered across the dominant structure and entangled with other concerns. For another example, certain software designs seem to be hard to capture more abstractly as software components. This is the case for most of the ‘Gang of Four’ (GoF) design patterns [Gamma et al., 1995], which cannot be expressed better than in terms of prose, pictures and prototypes.

Our first contribution is to show that, with the modern expressive type systems starting to appear in object-oriented languages, we can in fact capture (at least the code aspects of) the VISITOR design pattern [Gamma et al., 1995] as a generic and type-safe visitor software component. Moreover, it is possible to capture a number of variations on the pattern within one parametrizable component — specifically, we can support the following design decisions: who is responsible for traversing the object structure, the visitor or the object structure; whether the visitor is imperative (with results of traversals stored as mutable state in the visitor) or functional (pure, with results returned by the accept method); what orthogonal concerns such as tracing or caching of computations are supported. Instead of committing to a particular decision at the time we design a visitor, as would be necessary with the informally-expressed VISITOR pattern, we can define a single visitor that postpones all of these design decisions by allowing them to be specified by parametrization.

Our component is implemented in the Scala programming language [Odersky, 2006] and its type safety is statically guaranteed by the type system. The Scala features that make this possible are parametrization by type (or generics, as found in recent version of Java or C#) and abstract types (although type-constructor polymorphism [Altherr and Cremet, 2007] could be used instead). As far as we are aware, all existing solutions trying to capture some notion of generic visitors [Palsberg and Jay, 1998, Visser, 2001, Grothoff, 2003, Forax et al., 2005, Meyer and Arnout, 2006] make use of reflection or introspection mechanisms that do not statically guarantee type-safety. Furthermore, most of
those solutions only capture particular variations of the pattern.

Our second contribution is a semantics for a generalised algebraic datatype notation. The notation allow us to define parametric, mutually-recursive and existential visitors, being comparable in expressive power to Haskell 98 and ML-style datatypes. It also integrates well with object-oriented languages, allowing both datatypes and data-constructors to override or define new fields and methods. Furthermore, it generalises traditional algebraic datatypes, in the sense that both the traversal and the dispatching strategies are parametrizable.

2. The VISITOR as a Design Pattern
2.1 The VISITOR Pattern
The VISITOR design pattern is an alternative to the normal object-oriented approach to hierarchical structures, separating the operations from the object structure. Figure 1 shows the class structure of the pattern. The Visitor interface declares visit methods for each ConcreteElement type, implemented in each ConcreteVisitor class; the Element abstract superclass declares the accept method, taking a Visitor as argument, defined in each ConcreteElement subclass to select the appropriate visit method from a Visitor.

In contrast to the standard object-oriented decomposition, the VISITOR pattern makes it easy to add new operations — at the cost of making it difficult to add new variants. One can see the pattern as a way of simulating double dispatch in a single-dispatch language: the method implementation chosen depends on the dynamic types of both the ConcreteElement and the ConcreteVisitor.

2.2 Imperative and Functional VISITORS
In the traditional presentation of the visitor pattern, the visit and accept methods return no result; any value computed by the visitor is stored in the visitor for later retrieval. An alternative is for the visit and accept methods to return the value directly. Buchlovsky and Thielecke [2005] use the term imperative visitor for one that has visit and accept methods that return void, with all computations executed through side-effects, accumulating results via mutable state; in contrast, a functional visitor is immutable, all computations yielding their results through the return values of the visit and accept methods, which are pure.

2.3 Internal and External VISITORS
Gamma et al. [1995] raise the question of where to place the traversal code: in the object structure itself (in the accept methods), or in the concrete visitors (in the visit methods). Buchlovsky and Thielecke [2005] use the term internal visitor for the former approach, and external visitor for the latter. 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.

Figure 2 shows examples of the two variations, using functional-style VISITORS in Scala. In both visitors, the trait Tree and the classes Empty and Fork define a COMPOSITE. Using the visitor terminology, Tree is the element type and Empty and Fork are the concrete elements. The method accept, defined in Tree and implemented in the two concrete elements, takes a TreeVisitor object with two visit methods (one for each concrete element). 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 constructed object. Parametrizing the visit methods in this way gives a functional programming feel when using visitors.

Operations on trees are encapsulated in ConcreteVisitor objects. For example, an external visitor to compute the depth of a binary tree — explicitly propagating itself to subtrees — is defined as follows:

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

Defining values of type Tree benefits from Scala’s case class syntax, which avoids some 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, we define a method test to compute the depth of a small tree.

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

2.4 The Class Explosion
As is the case with most design patterns, the VISITOR pattern presents the programmer with a number of design decisions. An obvious dimension of variation follows the shape of the object structure being traversed: the Visitor interface for binary trees will differ from that for lists. We have just discussed two other dimensions of choice: imperative versus functional behaviour, and internal versus external control. A fourth dimension captures certain cross-cutting concerns, such as tracing of execution and memoization of results.

Handled naively, this flexibility introduces some problems. For one thing, capturing each combination separately leads to an explosion in the number of classes: ImpExtTreeBasicVisitor for imperative external tree visitors, FuncIntTraceListVisitor for functional internal tracing list visitors, and so on. Secondly, the dependency on user-supplied information (the shape of the object structure) prevents these classes from being provided in a library. Finally, because the variations have different interfaces, the choice between them has to be made early, and is difficult to change.

All three of these problems can be solved, by specifying the variation by parametrization. The main contribution
of this paper is the provision of a generic visitor component, parametrizable on each of these dimensions: shape (of object structure), result type (hence imperative versus functional), strategy (internal versus external), and concern (cross-cutting).

### 3. Programming with the Visitor Library

#### 3.1 A Datatype Notation for Visitors

Inspired by datatype declarations from functional programming languages, we introduce a succinct data-like notation as syntactic sugar for the actual visitor library in Scala, without compromising clarity and expressiveness. We present this notation informally in this section; a formal account is presented in Section 6.

Consider the following Haskell [Peyton Jones, 2003] datatype definition:

```haskell
data Tree =
    Empty
    | Fork Int Tree Tree
```

An equivalent definition in our data notation is:

```scala
data Tree {
    constructor Empty
    constructor Fork (x: int, l: Tree, r: Tree)
}
```

The following table presents the correspondence between the concepts in our visitor library and the traditional VISITOR pattern notation.

<table>
<thead>
<tr>
<th>Library notation</th>
<th>VISITOR terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>data T</td>
<td>Element</td>
</tr>
<tr>
<td>constructor</td>
<td>Concrete Element</td>
</tr>
<tr>
<td>(D)CaseT</td>
<td>Visitor</td>
</tr>
<tr>
<td>V extends (D)CaseT</td>
<td>Concrete Visitor</td>
</tr>
<tr>
<td>new (D)CaseT</td>
<td>Anonymous Concrete Visitor</td>
</tr>
</tbody>
</table>

The traits (D) CaseT are generated from the datatype definitions. For the tree example, this means that we would have DCaseTree and CaseTree traits.

**Generalized data notation** We also make our data notation more amenable to object-oriented programming by generalizing it so that datatypes can, in the same way as classes or traits, define and override methods and values. In Figure 3, we define a new Nat datatype that uses this generalized notation; it overrides the toString, equals and hashCode methods and defines a val intValue that is implemented by each of the constructors.
Figure 2. Internal and External VISITORS for Binary Trees

data Nat {
  val intValue : Int
  constructor Zero { val intValue = 0 }
  constructor Succ (n : Nat) { val intValue = 1 + n.intValue }
  override def toString () : String = this.accept {
    new CaseNat [Internal, String] {
      def Zero = "Zero"
      def Succ (n : String) = "Succ(" + n + ")"
    }
  }
  override def equals (x : Any) : boolean = x match {
    case m : Nat => intValue.equals (m.intValue)
    case _ => false
  }
  override def hashCode () = intValue.hashCode ()
}

External Visitors

trait Tree {
  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] {
  def empty : R
  def fork (x : int, l : R, r : R) : R
}

3.2 Traversal Strategies and the Functional Notation

While conventional datatypes normally use case analysis or pattern matching to decompose values, visitors have a choice of traversal strategies: internal, and external. Case analysis and pattern matching are a form of the latter. Consider, for example, a definition of the depth function on trees in Haskell:

```
data Nat {
  val intValue : Int
  constructor Zero { val intValue = 0 }
  constructor Succ (n : Nat) { val intValue = 1 + n.intValue }
  override def toString () : String = this.accept {
    new CaseNat [Internal, String] {
      def Zero = "Zero"
      def Succ (n : String) = "Succ(" + n + ")"
    }
  }
  override def equals (x : Any) : boolean = x match {
    case m : Nat => intValue.equals (m.intValue)
    case _ => false
  }
  override def hashCode () = intValue.hashCode ()
}

depth :: Tree -> Int
depth t = case t of
  Empty -> 0
  Fork x l r -> 1 + max (depth l) (depth r)
```

This corresponds, in our library, to:

```
def depth1 = new CaseTree [External, Int] {
  def Empty = 0
  def Fork (x : int, l : R [TreeVisitor], r : R [TreeVisitor]) =
    1 + max (l.accept (this), r.accept (this))
}
```

Here, depth1 defines a new anonymous concrete visitor on Tree using the CaseTree visitor trait. The External type argument of CaseTree selects the external traversal strategy, which allows the programmer to explicitly drive the traversal through the accept methods. The int type argument specifies the return type of the visit methods Empty and Fork. R [TreeVisitor] is a type dependent on the traversal strategy; in the case of external visitors, it is effectively a type synonym for the Tree composite\(^1\). For the remainder of the paper, for clarity, we will use the composite type directly instead for specifying the recursive types for external visitors.

**Functional Notation** Calling the accept method repeatedly is awkward. In Scala, functions are objects, so we can use a functional notation by making visitors a subclass of functions with composites as arguments. With this notation, depth1 can be rewritten as follows, which nicely reflects the recursive nature of the definition:

```
def depth2 = new CaseTree [External, Int] {
  def Empty = 0
}
```

\(^1\)Unfortunately, for external visitors, Scala does not allow us to write def (x : int.l : Tree, r : Tree) directly (we believe this may be a bug).
def Fork (x:int, l:Tree, r:Tree) =
    1 + max (depth2 (l), depth2 (r))
}

Internal Visitors In the definitions of depth1 and depth2, the particular traversal strategy used is parametrized on the concrete visitor instead of being fixed by the visitor component. This is a major advantage of our visitor library over the traditional design pattern interpretation: we do not need to commit in advance to a particular strategy when designing a new visitor. For example, instead of using external visitors to define the depth functions, we could have used instead an internal visitor:

```python
def depth3 = new CaseTree [Internal, int] {
    def Empty = 0
    def Fork (x:int, l:int, r:int) = 1 + max (l, r)
}
```

Since with internal visitors use traversal strategies determined by the elements, the above definition does not require explicit traversal of the structure, so it is simpler to define. In the case of internal visitors, R [TreeVisitor] is just a type synonym for int, which we use to give the types for l and r.

3.3 Advice and Modular Concerns

Having explicit control over traversal gives us the capability of decoupling non-functional concerns from base programs into localized modules and invoking them at each step of recursion. Inspired by Aspect-Oriented Programming (AOP) [Kiczales et al., 1997], we term such localized non-functional concerns advice\(^2\). Consider the following (naive) version of the Fibonacci function defined over Nat.

```python
def fib1 = new CaseNat [External, int] {
    def Zero = 0
    def Succ (n:Nat) = n.accept {
        new CaseNat [External, int] {
            def Zero = 1
            def Succ (m:Nat) = fib1 (n) + fib1 (m)
        }
    }
}
```

Though straightforward, the above definition has exponential time complexity. One way around this is memoization [Michie, 1968], which involves caching and reusing the computed results. Memoization is an orthogonal concern to the base computation, and cross-cuts [Kiczales et al., 1997] different functions, so is likely to become entangled with those functions.

Our visitor library offers a way to overthrow this ‘tyranny of the dominant decomposition’: it allows parametrization by dispatching, which can be used to introduce advice like memoization. In order to benefit from this additional power, we explicitly parametrize fib by the dispatching behaviour:

```python
def fib2 (d:Dispatcher [NatVisitor, External, int]) =
    new DCaseNat [External, int] (d) {
        def Zero = 0
        def Succ (n:Nat) = n.accept {
            new CaseNat [External, int] {
                def Zero = 1
                def Succ (m:Nat) = fib2 (d) (n) + fib2 (d) (m)
            }
        }
    }
```

Instead of CaseNat, we use the more general DCaseNat, a visitor parametrized by a Dispatcher (a type defined in our library, explained in detail in Section 5). The fib2 function now takes an extra value argument that determines dispatching and passes it to the constructor of DCaseNat. We include several commonly used pieces of advice in our library, and provide templates for user-defined new ones. We discuss a few of them below.

- **Basic** — the simple dispatcher, which defines the default behaviour of a visitor;
- **Memo** — memoization of results;
- **Advice** — a template for defining new dispatchers, which has before and after methods that are triggered before and after calls;
- **Trace** — tracing a computation by printing out the input and output, implemented using Advice as template.

More than one piece of advice can be deployed at the same time by composing them together. The special Basic dispatcher is atomic and is used as the unit of composition. Here are a few possible instantiations of fib2:

```python
def nfib = fib2 (Basic)
def mfib = fib2 (Memo (Basic))
def tmfibr = fib2 (Trace (Memo (Basic)))
def tmfib = fib2 (Memo (Trace (Basic)))
```

The program nfib is equivalent to fib1, while mfib is a version with memoization. The programs tmfibr and tmfib combine tracing and memoization in different ways and, while both programs return the same output for any given input, the trace written to the console is different. In our library, the ordering of advices is determined by the order of composition. In tmfibr, trace is triggered before Memo, which prints out all calls including those resorting to memoization. On the other hand, tmfib only prints out traces that do not involve memoization, as Memo (which can be seen as an around advice) takes precedence and may bypass the tracing.

3.4 Imperative Visitors

The GoF presentation of the VISITOR pattern discusses both internal and external imperative visitors; the emphasis is

---

\(^2\) In contrast to the pointcut mechanism in AOP, our advice is installed by parametrization. We leave a detailed comparison to Section 7.
on the internal variant, with external visitors being recommended for advanced uses (where the recursion scheme does not fit the internal variant). As it turns out, imperative visitors are a special case of functional visitors, with the return type set to void (or Unit, in Scala). For example, suppose we wanted to add all the integers in some tree, but we wanted to do so by using an imperative visitor that accumulates the value of the sum in a mutable variable. Using an internal visitor, we could write that program in Scala as:

```scala
class AddTree extends CaseTree [Internal, Unit] {
  var sumValue = 0
  def Empty = {}
  def Fork (x: int, l: Unit, r: Unit) = {
    this (l); this (r); sumValue += x;
  }
}
```

We could also write an imperative external version of the visitor as:

```scala
class AddTree2 extends CaseTree [External, Unit] {
  var sumValue = 0
  def Empty = {}
  def Fork (x: int, l: Tree, r: Tree) = {
    this (l); this (r); sumValue += x;
  }
}
```

In this case, we need to explicitly traverse the structure, by applying the visitor to the composites (remember that `this (l)` is `l.accept (this)`). The imperative visitors are used as follows:

```scala
def test : int = {
  val addTree = new AddTree2();
  val tree1 = Fork (3, Empty (), Empty ());
  val tree2 = Fork (4, tree1, tree1);
  addTree (tree2);
  return addTree.sumValue;
}
```

Here, `AddTree2` should be replaced by either `AddTree1` or `AddTree2`. The program creates a new instance `addTree` of `AddTree2`, defines the value `tree2`, applies `addTree` to it, then returns the value accumulated by the visitor traversal in the variable `sumValue`.

### 3.5 A Simple Form of Multiple Dispatching

As we mentioned in Section 2.1, the VISITOR pattern simulates double dispatching in a single-dispatching language. The use of nested external visitors allows us to go further, and simulate multiple dispatching. For example, we could define a type-safe (in the sense that no casts are required) equality function by using this nesting technique. Figure 4 shows an implementation; the method takes two trees as arguments, performs a case analysis (using an external visitor) on one of the trees, then in both the `Empty` and `Fork` cases, performs a case analysis on the other tree.

Note that this version of equality requires triple dispatching, because the method is defined in some object `A`, which is used to dynamically determine the implementation of `equal`, and the two tree arguments need to be dynamically inspected. We could, of course, have defined a version of equality that would only require double dispatching, by placing the method `equal` in `Tree` and taking another `Tree` as an argument.

While this technique can be used to emulate a form of multiple dispatching, the programs start suffering from readability issues, due to the nesting of visitors. Similar problems occur in functional programming languages, when multiple nested case analyses are used. To alleviate these, many of those languages introduce pattern matching as syntactic sugar on top of case analysis, allowing a definition like `equal` to be written as follows:

```scala
def isEmpty = new CaseTree [External, boolean] {
  def Empty = true
  def Fork (x: int, l: Tree, r: Tree) = false
}
def equal (t: Tree) : Tree => boolean = new CaseTree [External, boolean] {
  def Empty = isEmpty (t)
  def Fork (x: int, l: Tree, r: Tree) = t.accept [External, boolean] (
    new CaseTree [External, boolean] {
      def Empty = false
      def Fork (y: int, l2: Tree, r2: Tree) = { x ≡ y ∧ equal (l1) (l2) ∧ equal (r1) (r2) }
    }
  )
}
```

Figure 4. A type-safe equality function using External Visitors.

### 3.6 Parametrized and Mutually Recursive Visitors

The expressiveness of our library extends to parametrized and mutually recursive visitors. An example is forests and trees:
data $\text{Tree}[a]$
  \begin{align*}
  &\text{constructor} \text{Fork} \ (x : a, f : \text{Forest}[a]) \\
  &\text{constructor} \text{Nil} \\
  &\text{constructor} \text{Cons} \ (t : \text{Tree}[a], f : \text{Forest}[a])
  \end{align*}

Trees, of type $\text{Tree}[a]$, have one constructor $\text{Fork}$ that builds
a tree containing one element of type $a$ and a forest; forests,
of type $\text{Forest}[a]$, have two constructors $\text{Nil}$ and $\text{Cons}$ that
construct empty and non-empty forests.

We could define a function to sum all the leaves of a tree
of integers as follows:
\[
def \text{sumTree} = \text{new CaseTree} [\text{Internal}, \text{int}, \text{int}] \{
  \text{def} \text{mrefForest} = \text{sumForest} \\
  \text{def} \text{fork} \ (x : \text{int}, xs : \text{int}) = x + xs \\
  \text{def} \text{sumForest} = \text{new CaseForest} [\text{Internal}, \text{int}, \text{int}] \{
  \text{def} \text{mrefTree} = \text{sumTree} \\
  \text{def} \text{nil} = 0 \\
  \text{def} \text{cons} \ (x : \text{int}, xs : \text{int}) = x + xs
  \}
\]

Due to the mutually dependent nature of the two visitors,
a function that traverses one must know of a corresponding
function on the other. For this reason, mutually recursive vis-
itors contain fields referring to the visitors that they depend
on. We name such fields $\text{mrefForest}$ and $\text{mrefTree}$ (the details
are explained in Section 6). Additionally, for parametrized
types like $\text{Tree}[a]$, type arguments (such as $a$) are also passed
as arguments to $\text{CaseTree}$.

4. Visitors as Encodings of Datatypes

4.1 Encoding Datatypes in the Lambda Calculus

In the pure lambda calculus, there is no native notion of
datatype; this has to be encoded using functions. Church
[1936] showed how to encode the natural numbers via re-
peated function composition: the number 0 is represented by
‘zero-fold composition’, the number 1 by ‘one-fold composi-
tion’, the number 2 by ‘two-fold composition’, and so on.
\[
\begin{align*}
  \text{zero} &\equiv \lambda f \Rightarrow \lambda x \Rightarrow x \\
  \text{succ} &\equiv \lambda n \Rightarrow \lambda f \Rightarrow \lambda x \Rightarrow f \ (n \ f \ x)
\end{align*}
\]

Much later, Böhm and Berarducci [1985] demonstrated
precise typings of such encodings in System F. The name
‘Church encoding’ is normally associated with Böhm and
Berarducci’s System F encoding. Church encodings allow us
to write iterative definitions. A less well-known encoding
is the Parigot encoding [Parigot, 1992], which allows us to
write recursive definitions, but requires System F to be
extended with recursion. Splawski and Urzyczyn [1999] give
precise definitions of iteration versus recursion in this sense;
we shall not dig into the details in this paper.

Figure 5 shows the Church and Parigot encodings of
naturals and trees in a System F-like calculus extended with
recursion. For Church encodings, the types $\text{Nat}$ and $\text{Tree}$
are not recursive: the constructors traverse the structure,
and the functions that form the basis of those two types only
need to process the results of those traversals. In contrast,
with Parigot encodings, the constructors do not traverse the
structure; therefore, the functions that represent $\text{Nat}$ and $\text{Tree}$
need to define the traversal themselves. This requires that the
types of those functions recursively refer to $\text{Tree}$ and $\text{Nat}$,
which can only be achieved if we allow recursive types. Note
that the internal and external visitors presented in Figure 2
 correspond very closely to, respectively, the Church and
Parigot encodings for trees (although we use $\text{Nat}$ instead of
$\text{int}$ here).

4.2 Generic Visitors: Shape Abstraction

We are not the first to realize that visitors are related to
encodings of datatypes; in fact, it has become folklore knowl-
edge among some communities. Buchlovsky and Thielecke
[2005], in work directed to the type-theory community,
formalized the relation between visitors and encodings of
datatypes precisely and showed a single shape-generic form
of the encodings.

The traditional presentation of encodings of datatypes in
System F (and common variants) [Girard et al., 1989] is of
the form:
\[
T \equiv \forall \Sigma. \ (F \ R \Rightarrow X) \Rightarrow X
\]

where the operation on types $F$ specifies the shape of the
datatype. Typically, $F \ R$ takes the form of a sum of products
$\Sigma_i F_i \ R$, a collection of variants in which each $F_i \ R$ is a simple
product of types; so the encoding is equivalent to
\[
T \equiv \forall \Sigma. \ ((\Sigma_i F_i \ R) \Rightarrow X) \Rightarrow X
\]

Now, the type $(\Sigma_i F_i \ R) \Rightarrow X$ of functions from a sum is iso-
morphic to the type $\Pi_i (F_i \ R \Rightarrow X)$ of products of functions
(in the same way that $x^{n+z} = x^n \times x^z$); so another equivalent
encoding is:
\[
T \equiv \forall X. \ (\Pi_i (F_i \ R \Rightarrow X)) \Rightarrow X
\]

Buchlovsky and Thielecke [2005] point out that this clearly
relates the datatype $T$ with the type of its accept method
$\forall X. \ (\Pi_i (F_i \ R \Rightarrow X)) \Rightarrow X$: the latter can be read, for some
result type $X$, as taking a visitor of type $\Pi_i (F_i \ R \Rightarrow X)$
and yielding a result of type $X$; the visitor itself is just a
collection of functions of the form $F_i \ R \Rightarrow X$, each being the
visit method for one variant of the datatype, with argument
vector $F_i \ R$.

Church and Parigot encodings — corresponding, respec-
tively, to internal and external visitors — follow from two
specific instantiations of $R$. For reference, define operation
$V$ by $V \ R \ X \equiv \Pi_i (F_i \ R \Rightarrow X)$.

- *Generic internal visitors* are obtained by specializing $R \equiv
X$; we can define
Ever, this definition is not valid in System F, because it is still not right. To see what the problem is, let’s first reformulate the Church peano numerals using products of functions, as in Figure 6. When we try to use Composite NatF instead of Internal NatF:

\[
\text{Nat} \equiv \text{Composite NatF}
\]

\[
\text{zero} \equiv \lambda s \, z \Rightarrow z
\]

\[
\text{succ } n \equiv s \, z \Rightarrow s \, (n \, s 
\]

\[
\text{Tree} \equiv \forall A. \, A \Rightarrow (\text{Nat} \Rightarrow A) \Rightarrow A
\]

\[
\text{empty} \in \text{Tree}
\]

\[
\text{empty} \equiv \lambda e \, f \Rightarrow e
\]

\[
\text{fork} \in \text{Nat} \Rightarrow \text{Tree} \Rightarrow \text{Tree} \Rightarrow \text{Tree}
\]

\[
\text{fork} \, x \, l \, r \equiv \lambda e \, f \Rightarrow f \, x \, (l \, e \, f) \, (r \, e \, f)
\]

\[
\text{NatF} \Rightarrow \text{RA} \equiv (A, R \Rightarrow A)
\]

\[
\text{Nat} \equiv \text{Internal NatF}
\]

\[
\text{zero} \in \text{Nat}
\]

\[
\text{zero} \equiv \lambda (z, s) \Rightarrow z
\]

\[
\text{succ} \in \text{Nat} \Rightarrow \text{Nat}
\]

\[
\text{succ } n \equiv \lambda (z, s) \Rightarrow s \, (n \, (z, s))
\]

\[
\text{NatF} \Rightarrow \text{RA} \equiv (A, R \Rightarrow A)
\]

\[
\text{zero} \in \text{Nat}
\]

\[
\text{zero} \equiv \lambda s \, z \Rightarrow z
\]

\[
\text{succ} \in \text{Nat} \Rightarrow \text{Nat}
\]

\[
\text{succ } n \equiv \lambda s \, z \Rightarrow s \, n
\]

\[
\text{Tree} \equiv \forall A. \, A \Rightarrow (\text{Nat} \Rightarrow \text{Tree} \Rightarrow \text{Tree} \Rightarrow A) \Rightarrow A
\]

\[
\text{empty} \in \text{Tree}
\]

\[
\text{empty} \equiv \lambda e \, f \Rightarrow e
\]

\[
\text{fork} \in \text{Nat} \Rightarrow \text{Tree} \Rightarrow \text{Tree} \Rightarrow \text{Tree}
\]

\[
\text{fork} \, x \, l \, r \equiv \lambda e \, f \Rightarrow f \, x \, (l \, e \, f) \, (r \, e \, f)
\]
Note that the Decompose $S$ parameter is implicitly passed.

In order to define new strategies, we need to define some concrete type $S$ and the corresponding $\text{dec}_S$ operation. For example, to make internal and external visitors two instances of $\text{Composite} V$, we specialize $S$ to $\text{Internal}$ and $\text{External}$: 

\[ \text{Internal} V X \equiv X \]

\[ \text{External} V X \equiv \text{Composite} V \]

Here we reuse the identifiers $\text{Internal}$ and $\text{External}$ to refer to the associated traversal strategies. The specific instantiations of $\text{dec}_S$ for internal and external visitors are:

\[ \text{dec}_{\text{Internal}} \in (V (\text{Internal} V X) X) \Rightarrow \text{Composite} V \Rightarrow \text{Internal} V X \]

\[ \text{dec}_{\text{External}} \in (V (\text{External} V X) X) \Rightarrow \text{Composite} V \Rightarrow \text{External} V X \]

In the definition of $\text{dec}_{\text{Internal}}$ the reader should (again) note that the $\text{Decompose} S$ parameter is implicitly passed and, therefore, the composite $c$ just needs to take the visitor $v$ as an argument. With $\text{dec}_{\text{External}}$, we simply ignore the visitor parameter and return the composite itself. This allows the use of the composite directly in the definitions of the visit methods.

5. A Scala Library for Visitors

In the previous section, we used the Church and Parigot encodings of datatypes to motivate a notion of visitors that is 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.

We use the results from Section 4.3 as a functional specification for our Scala visitor library. The translation from the functional specification into a Scala component is relatively straightforward, although some typings vary slightly due to the differences between System F-like languages and Scala. We start by recalling the definition of $\text{Composite}$, and annotate it with extra information identifying the accept method and the visitor component.

\[ \text{Composite} V \equiv \forall X S. \text{Decompose} S \Rightarrow V (S V X) X \Rightarrow X \]

In order to implement the different components present in the functional specification we will make extensive use of generics (parametrization by types) and abstract types [Odersky, 2006], which provide a means to abstract over concrete types used inside a class or 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]. They are considered by Odersky and Zenger [2005] to be essential for the construction of reusable components; they allow information hiding over several objects, a key part of component-oriented programming [Pflister and Szyperski, 1996].

Alternatively to abstract types, we could have used type-constructor polymorphism [Altherr and Cremer, 2007] instead. A Haskell solution that exploits this approach is shown in Oliveira [2007]. Since Scala now supports type- constructor polymorphism [Moors et al., 2007], a solution using such an approach should also be possible. However, as discussed by Oliveira, abstract types seem to be more expressive than type-constructor polymorphism alone, and allow the definition of a slightly more general visitor library.

Visitors and the Functional Notation The Visitor component in the library, which captures the shape of the type $V$ in the functional specification, has two abstract types: $S$ (representing the traversal strategy) and $X$ (representing the return type of the visitor). The Visitor also contains a type $R$ that corresponds to the type $S V X$ (the first argument of $V$, specifying the type of recursive arguments).

\[
\text{trait Visitor} \{ \\
\text{type } X \\
\text{type } S \triangleq \text{Strategy} \\
\text{type } R[v <: \text{Visitor}] = (S \{\text{type } V = \text{Visitor}.\text{this}.X; \text{type } V = v\}) \# Y \\
\}
\]

The notation $T \# Y$ used in the definition of the type synonym $R$ is the equivalence of obj-method on type level. In other words, $T \# Y$ selects the type $Y$ from the trait or class $T$. We will explain the type $Y$ when we introduce Strategy.

We also introduce a type synonym $\text{VisFunc}$ parametrized by a visitor $v$, a strategy $s$ and a result type $x$, as a shortcut for visitors that are also functions.

\[
\text{type VisFunc}[v <: \text{Visitor}, s <: \text{Strategy}, x] = \\
\text{Function1}[\text{Composite}[v], x] \text{ with } \\
v \{\text{type } S = s; \text{type } X = x\}
\]

In essence, we treat visitors as functions that take a $\text{Composite}[v]$ as an argument and return a value of type $x$, by observing that the invocation $a.\text{accept}(f)$ where $a$ is a composite and $f$ is a visitor can be interpreted as a form of function application $f(a)$. The with keyword is used in Scala to do mixin composition of traits.

Composites The $\text{Composite}$ 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 traversal strategy to use while visiting the structure.

\[
\text{trait Composite}[v <: \text{Visitor}] \{ \\
\text{def } \text{accept}[s <: \text{Strategy}, x] (\text{vis} : \text{VisFunc}[v, s, x]) \\
\text{(implicit } \text{decompose} : \text{Decompose}[s] : x \}
\]
We switch the order of the two arguments (when compared to the Composite equation shown earlier) because `decompose` can be implicitly inferred (since it is determined by the concrete instantiation of `s`), and Scala requires implicit arguments to be placed last.

**Traversals Strategies** The shape of the parameter `S` is captured in Scala by the following trait:

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

A `Strategy` has two abstract types `V` and `X` and a type `Y` that is dependent on `V` and `X` (although that dependency is not captured directly by Scala’s type system). The type `Y` represents the type used in place of recursive occurrences in the `visit` methods. Subtypes of this trait will correspond to different possible traversal strategies for the visitors. In particular, the strategies `Internal` and `External` are defined as:

```scala
trait Internal extends Strategy {
  type Y = X
}
```

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

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

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

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

```scala
implicit def internal : Decompose[Internal] =
  new Decompose[Internal] {
    def dec[v <: Visitor, x] (vis : VisFunc[v, Internal, x],
      comp : Composite[v]) = vis.apply (comp)
  }

implicit def external : Decompose[External] =
  new Decompose[External] {

    def dec[v <: Visitor, x] (vis : VisFunc[v, External, x],
      comp : Composite[v]) = 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 `dec`, which is `vis.apply (comp)` for internal visitors and just `comp` for external visitors. In essence, the traversal strategy of the internal visitors recurs on the composite `comp` (since it calls the `accept` method via `apply`); and the traversal strategy for external visitors returns the composite untouched, which allows concrete visitors to control recursion themselves.

**Dispatchers** In Scala, functions are not primitive: they are defined as a trait `Function1` with an `apply` method. This means that we can provide our own implementation of the `apply` methods, which allows us to add extra behaviour on function calls. Our visitor library has the notion of a dispatcher, allowing us to parametrize the dispatching behaviour of our visitors, adding an extra form of parametrization that is not considered by the functional specification.

Figure 7 shows the trait that defines the interface of a `Dispatcher` and a few implementations of that trait. The method `dispatch` takes a visitor and a traversal strategy and returns a function that will be used by the `apply` method in the visitor to define the dispatching behaviour. The definition `Basic` implements `Dispatcher` with the standard dispatching behaviour by just calling the `accept` method. The class `Advice`, inspired by the notion of advice in AOP, wraps itself around another dispatcher and defines `dispatch` as a `TEMPLATE METHOD` [Gamma et al., 1995] that calls the `before` and `after` methods around the `dispatch` function of the dispatcher argument. One implementation of advice is given by `Trace`, which provides a simple tracing concern that prints the arguments before performing a call and prints the result after returning. Finally, the `Memo` dispatcher implements a form of memoization: it intercepts function calls so that only calls on values that have not been seen before are performed — results for other calls are retrieved from a cache.

We should emphasize that dispatchers are composable having `Basic` as the unit of composition. Furthermore, new ones can be easily added.

**The Case Visitor** Having built the basic building blocks for the visitor library, we now introduce the `Case` class, which will be used to provide the functional notation and to define concrete visitors:

```scala
abstract class Case [v <: Visitor, s <: Strategy, x] {
  (d : Dispatcher [v, s, x]) (implicit dec : Decompose [s])
  extends Function1 [Composite[v], x] {
    self : Case [v, s, x] with v {type S = s; type X = x} ⇒
```
The class `Case` extends `Function1` and the `apply` method is defined by calling the `dispatch` method from the provided dispatcher `d`.

6. Translation of Datatypes

In this section we define a translation scheme between datatype-like declarations and visitors defined using our Scala library. We introduce a mini-language for datatypes as follows.

```
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Scala Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>τ ::= data T [α] = {e₁, e₂}</code></td>
<td><code>data T [α] = {e₁, e₂}</code></td>
</tr>
<tr>
<td><code>c ::= constructor X [β] :: T [α]</code></td>
<td><code>constructor X [β] :: T [α]</code></td>
</tr>
<tr>
<td>`t ::= T₁ [t₁]</td>
<td>T₀ [α]`</td>
</tr>
<tr>
<td><code>Non-recursive Types</code></td>
<td>`α</td>
</tr>
<tr>
<td><code>Types</code></td>
<td>`α</td>
</tr>
<tr>
<td><code>s ::= Scala declarations</code></td>
<td><code>Scala declarations</code></td>
</tr>
</tbody>
</table>
```

A datatype `τ`, possibly parametrized by type variables `α`, introduces a set of data constructors and some optional Scala code `s`. Each constructor, `X [β]`, can take an optional list of type arguments `β`.

The class `Case` is type-parametrized by a visitor `v` (the shape argument), a strategy `s` (the traversal strategy argument) and the return type `x`. Furthermore, it is also value-parametrized by a `d` (the dispatcher argument) and an implicit value `dec` (related to the traversal strategy). Subclasses of `Case` will implement the visitor type `v` passed as an argument. This is expressed by Scala’s `self-type` annotation `self: Case[v, s, x] with v {type S = x; type X = x}`. The class `Case` extends `Function1` and the `apply` method is defined by calling the `dispatch` method from the provided dispatcher `d`.

6. Translation of Datatypes

In this section we define a translation scheme between datatype-like declarations and visitors defined using our Scala library. We introduce a mini-language for datatypes as follows.

```
<table>
<thead>
<tr>
<th>Syntax</th>
<th>Scala Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>τ ::= data T [α] = {e₁, e₂}</code></td>
<td><code>data T [α] = {e₁, e₂}</code></td>
</tr>
<tr>
<td><code>c ::= constructor X [β] :: T [α]</code></td>
<td><code>constructor X [β] :: T [α]</code></td>
</tr>
<tr>
<td>`t ::= T₁ [t₁]</td>
<td>T₀ [α]`</td>
</tr>
<tr>
<td><code>Non-recursive Types</code></td>
<td>`α</td>
</tr>
<tr>
<td><code>Types</code></td>
<td>`α</td>
</tr>
<tr>
<td><code>s ::= Scala declarations</code></td>
<td><code>Scala declarations</code></td>
</tr>
</tbody>
</table>
```

A datatype `τ`, possibly parametrized by type variables `α`, introduces a set of data constructors and some optional Scala code `s`. Each constructor, `X [β]`, can take an optional list of type arguments `β`.
arguments \( \hat{\rho} \) (which act as existentially quantified types) and a list of labelled type arguments \( \forall \tau T \). Scala definitions \( \rho \) can be inserted to define or override fields and methods. We single out non-recursive type arguments, \( t_i \), which do not make self-reference to the datatype to introduced them. Recursive occurrences of type constructors are denoted \( \varnothing_0 \), to separate them from the non-recursive ones (\( \forall \tau T \)).

The reason for this separation is to enforce a few syntactic restrictions on the language. In particular, nested datatypes [Bird and Meertens, 1998] and constructors with functional parameters having recursive occurrences [Meijer and Hutton, 1995] are excluded, since traversals are hard to define for those types. Despite these restrictions, the \textbf{data} constructor presented here is comparable in expressive power to ML-style and Haskell 98-style datatypes, allowing us to express \textit{(type-)parametrized datatypes, mutually recursive datatypes and existential datatypes.}

Declarations in the datatype language can be translated to visitors by the meta-function \textit{GEN} in Figure 8. Before going into the details of the translation, we first introduce a few notational conventions. We write \( \alpha \) for a sequence of entities numbered from 1 to \( n \) and \( \alpha_i \) as the \( \alpha^b \) of them. We use a pattern matching syntax \( t_i @ \tau_1 [\tau_2] \) to denote that the bound variable \( t \) is of type \( \tau_1 \), \( \tau_2 \) for some \( \tau_1 \) and \( \gamma \). New names for visitors and references are created by prefixing or postfixing with the type constructor name, for example \( \tau \text{Visitor} \). We assume a dependency analysis and write \( \tau_m \) to denote the set of mutually recursive types that \( \tau \) makes references to (excluding \( \tau \) itself).

For each datatype \( \tau \), we generate a corresponding visitor type (a trait that extends Visitor) and a composite (a trait that extends Composite \([\tau \text{Visitor}] [\alpha] \)). We also generate two auxiliary visitors \( \text{DCase}_\tau \) and \( \text{Case}_\tau \). The former extends Case \([\tau \text{Visitor}] [\alpha] , s , x \), providing a convenient way to parametrize visitors by traversal and dispatching strategies as well as allowing visitors to be interpreted as functions. The latter provides a shorthand for the Basic dispatching strategy. The function \textit{GENDATA} creates a case class for each constructor \( \chi \) extending \( \tau [\alpha] \) and generates the corresponding accept method by checking the recursive status of \( \chi \)’s arguments, which determines the traversal code.

Each of the visitors \( \tau \text{Visitor} [\alpha] \) may have mutually recursive references to other visitors that it depends on, which are generated by the function \textit{GENREF}. The types of the \textit{vis} methods \( \chi \) (named after the corresponding constructor) also depend on the recursive status of the constructor’s arguments and are generated by the \textit{GENTYPE} function.

In Figure 9 we apply the translation to the trees and forests example in Section 3.6. For the datatype \( \text{Tree}[a] \), we generate the visitor and composite types \( \text{TreeVisitor}[a] \) and \( \text{Tree}[a] \), the two auxiliary visitors \( \text{CaseTree} \) and \( \text{DCaseTree} \), and the constructor \( \text{Fork} \). The mutual dependency with \( \text{Forest}[a] \) is captured by the \textit{mrefForest} definition on \( \text{TreeVisitor}[a] \).

A similar process happens for \( \text{Forest}[a] \), resulting in the generation of \textit{ForestVisitor}[\alpha], \textit{Forest}[\alpha], \textit{CaseForest}, \textit{DCaseForest}, \textit{Nil} and \textit{Cons}. A mutual reference \textit{mrefTree} is also placed in \textit{ForestVisitor}[\alpha].

7. Discussion and Related Work

7.1 Traversal Strategies and Recursion Patterns

Traversal strategies are closely related to the recursion patterns studied by the Algebra of Programming movement [Bird and De Moor, 1997]. This work supports Hoare’s observation that data structure determines program structure; the shape of the data induces for free a number of patterns of computation, together with reasoning principles for those patterns.

The most familiar of these families of recursion patterns is the ‘fold’ (or ‘catamorphism’) operation, which performs structurally inductive computations reducing a term to a value. Better still, those similar definitions are related parametrically, and can all be subsumed in one single datatype-generic definition, parametrised by the shape. The internal visitors expressible with our library are basically folds.

The Algebra of Programming patterns can provide inspiration for new types of visitor, beyond what is well-known in the literature. For example, Meertens [1992] introduces the notion of a paramorphism, which in a precise technical sense is to primitive recursion what catamorphism is to iteration. Informally, the body of a paramorphism has direct access to the original subterms of a term, in addition to the results of traversing those subterms as a catamorphism does. The obvious definition of factorial, in which \( (n+1)! \) depends on \( n \) as well as \( n! \), is a representative application. This recursion pattern can be expressed as a strategy using our visitor library:

\begin{verbatim}
trait Para extends Strategy {
    type Y = Pair[X, Composite[V]]
}

implicit def para: Decompose[Para] =
    new Decompose[Para] {
        def dec[v <: Visitor,x] (vis : VisFunc[v,Para,x],
            comp : Composite[v]) =
            Pair[x, Composite[v]] (vis.apply (comp),comp)
    }
\end{verbatim}

7.2 Dispatching Strategies and Modular Concerns

Kiczales et al. [1997]’s aspect-oriented programming (AOP) aims at modularizing concerns that cut across the components of a software system. These ideas inspired some of the applications of our library in Section 3. In AOP, programmers are able to modularize these crosscutting concerns within locally defined aspects: pointcuts designate when and where to crosscut other modules, and advice specifies what will happen when a pointcut is reached. Although AOP successfully separates concerns that are scattered and tangled
throughout the program, it can also introduce a form of tight coupling between base programs and their aspects, which complicates modular program understanding and reasoning.

Several authors [Aldrich, 2005, Kiczales and Mezini, 2005, Gudmundson and Kiczales, 2001] have proposed ways to harness the power of aspects by giving more control to programmers over which parts of their code are open to advice. Notable among these are Aldrich’s open modules, which encapsulate function definitions into modules and export public interfaces for both calling and advising from other modules. Internal function calls that are private to a module can only be advised if the module explicitly chooses to allow this. In this sense, our use of advice through visitors is akin to the internal advising of open modules. Functions or modules that are subject to advice are parametrized by dispatchers and instantiated to a particular generic advice. A significant difference between our approach and open modules lies in the means of triggering advice: parametrization versus pointcuts. It is no surprise that our library does not have fully fledged support for AOP; however, a significant class of applications of AOP can be coded up conveniently and modularly.

7.3 Case Classes and Algebraic Datatypes

The datatype notation that we have introduced in this paper is inspired by algebraic datatypes (AlgDts) from functional programming. Scala [Odersky, 2006] has its own notion of AlgDts via (sealed) case classes. With case classes, we could rewrite the Tree and depth examples as:

```scala
sealed case class Tree
  case class Empty extends Tree
  case class Fork (x: int, l: Tree, r: Tree) extends Tree

  def depth (t: Tree): int = t match {
    case Empty () ⇒ 0
    case Fork (x, l, r) ⇒ 1 + max (depth (l), depth (r))
  }
```

The sealed keyword guarantees that the class hierarchy will not be extended in other modules. Sealing allows the Scala compiler to perform an exhaustiveness check, guaranteeing that an operation is defined for all cases. This gives us essentially the same advantages (and disadvantages) as AlgDts. However, simple case classes are more general than AlgDts, because they do not need to be sealed; we could have defined Tree without the sealed keyword, gaining the ability to add...
The possibility of new variants in the future. Nevertheless, this extra generality can create problems because, although new variants can be added, functions defined by matching cannot be extended, and exhaustiveness checks become unavailable, essentially introducing the possibility of “Message not understood” runtime errors.

There are three main differences between the notion of datatypes introduced in this paper and case classes. Firstly, AlgDts and case classes correspond, essentially, to visitors with traversal and dispatching strategies set to External and Basic, therefore losing much of the reusability benefits offered by those parametrizations. Secondly, although the datatype notation requires a language extension, the approach we have taken is mostly library-based. This has the important advantage that we can functionally provide the visitor library, without extending the compiler itself. For example, as we have seen in Section 7.1, it is very simple to add a new kind of traversal strategy. We believe that an approach could be taken similar to the one with Iterators [Gamma et al., 1995] in C# and new versions of Java, with a library component and some built-in language support (the foreach keyword). We envision a language extension supporting the datatype notation, perhaps also with a parametrizable case construct and pattern matching notation, built on top of the visitor library. Finally, the semantics of case classes is essentially given by type inspection and downcasting. Our semantics does not rely on the availability of casts or run-time type information, so it could be used in object-oriented languages without these mechanisms.

7.4 Generic Visitors

There have been several proposals for generic visitors (visitor libraries that can be reused for developing software using
something like 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 this idea, mostly to improve performance, have been proposed since by Grothoff [2003] and Forax et al. [2005]. Meyer and Arnout [2006] also present a generic visitor along the same lines, but having less dependence on introspection mechanisms (although those are still needed). One advantage of these approaches is that they are not invasive — that is, the visitable class hierarchies do not need to have accept methods and it is possible to write generic traversal code (i.e. code that works for different visitors). In this paper, we can avoid most of the direct uses of the accept methods by using the datatype and functional notations, but the methods will still be needed. Although we do not address the issue here, very flexible and type-safe generic traversal code can be written using a datatype-generic programming extension to our visitor library [Oliveira, 2007]. A disadvantage of introspection-based approaches is that they cannot statically ensure type-safety, and so strictly speaking should not be classified as components. Furthermore, those approaches lack flexibility in the choice of the dispatching policy [Cunei and Vitek, 2005].

Visser [2001] observes that the VISITOR pattern suffers from two main limitations: lack of traversal control; and resistance to combination, which are closely related to our notions of traversal and dispatching parametrization. His solution for those problems consists of a number of generic visitor combinators for traversal control. These combinators can express interesting traversal strategies like bottom-up, top-down or sequential composition of visitors and can be used to define visitor-independent (or generic) functionality. Like all other implementations of forms of generic visitors, Visser’s solution requires run-time introspection. It would be interesting to explore some of Visser’s ideas in the context of our visitor library.

7.5 Multiple Dispatch

Mainstream object-oriented languages, like C++, C# and Java, all use a single dispatching mechanism, where a single argument (the self object) is dynamically dispatched and all other dispatching is static. A problem arises, however, when a method requires dynamic dispatching on two or more arguments. There is a rich literature motivating and proposing solutions for this problem: Chambers and Leavens [1995], Clifton et al. [2000], Ernst et al. [1998] are just a few examples. Still, none of those solutions have been adopted by mainstream programming languages. Two reasons for this are the difficulty of providing modular (compile-time) type-checking to catch ambiguous and invalid combinations of dynamically dispatched arguments, and fears that multi-methods go against object-oriented principles like encapsulation.

Visitors can be used to emulate a (limited) form of multiple dispatching in object-oriented languages, as we mentioned in Section 2.1. Ambiguous and invalid combinations of dynamically dispatched arguments do not pose a problem for our visitors, but the price to pay for this is that we lose the ability to (easily) add new variants, which is possible with many of the multiple-dispatching solutions. Encapsulation is more problematic, and visitor-based solutions are often criticized as not being very object oriented. We agree that the idea of encapsulation is important and, whenever possible, it should be preserved. Nevertheless, for some problems a functional decomposition style is more appropriate, and trying to preserve (full) encapsulation is hard. What seems clearly worse to us than the loss of encapsulation is the fact that most object-oriented languages do not have an easy-to-use mechanism for a form of multiple dispatching (even if limited) except via the (statically) type-unsafe instanceOf introspection mechanism. We believe that our datatype notation and the related External traversal strategy could provide an easy-to-use and lightweight (if simple-minded) solution for the multiple dispatching problem.

8. Conclusions

We have argued that (the code aspects of) the VISITOR design pattern can be captured as a reusable, generic and statically type-safe component by using some advanced type system features that are starting to appear in modern object-oriented languages. We have shown that, inspired by functional programming, we can significantly improve the use of visitors by using a datatype-like and functional notations, while at the same time providing a simple functional decomposition mechanism that, we think, is well-suited for object-oriented languages.

This work is based on Chapter 3 of Oliveira [2007] dissertation that, in essence, builds on the insights provided by type-theoretic encodings of datatypes to derive a visitor software component. Other chapters of that dissertation address two other issues, related to visitors, not discussed here: datatype-generic programming (the ability to write functions that work for any visitors); and extensibility (the ability to add new variants to visitors). Solutions for those are also achieved without compromising static type-safety.

The hope is that this line of work will, more generally, show how more expressive forms of parametrization can help in resolving limitations of current programming languages when it comes to componentization and modularization of software. For the future, we would like to:

- Investigate possible programming languages extensions for the datatype-notation, as well as a case analysis and a pattern matching notation, with full support for all the parametrization aspects of the visitor library.
- Develop a formal setting that can be used to formalize and reason about components. In particular, we would
like to create a simple, but expressive, purely functional object-oriented language and investigate the implementation of some design patterns as components.

The code for this paper can be obtained from www.comlab.ox.ac.uk/people/Bruno.Oliveira/VisLib.tgz.

References


E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


