# 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.oliveira,meng.wang,jeremy.gibbons}@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 as anything more formal than prose, pictures and prototypes.

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 and modularly*: no reflection or similar mechanisms are used and modules can be independently compiled. The library is *generic*, in two senses: not only is it parametrised by both the return type and the shape of the object hierarchy, but also it allows a number of implementation choices (internal versus external control, imperative versus functional behaviour, orthogonal aspects such as tracing and memoisation) to be specified by parameters rather than fixed in early design decisions. Finally, we propose a generalised *datatype*-like notation, on top of our visitor library: this provides a convenient functional decomposition style in object-oriented languages.

***Categories and Subject Descriptors*** D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.3.2 [*Programming Languages*]: Language Classifications—Object-oriented languages, Applicative (functional) languages, Multiparadigm languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Recursion, Patterns, Data types and structures

***General Terms*** Languages, Design

***Keywords*** Design Patterns, Visitor Pattern, Software Components, Program Extensibility, Algebraic Datatypes, Traversal



*OOPSLA'08* October 19–23, 2008, Nashville, Tennessee, USA.


## 1. Introduction

A *software component* is, generally speaking, a piece of software that can be *safely reused* and *flexibly adapted* by some other piece of software. Safety can be ensured, for example, by a type system that guarantees correct usage; flexibility stems from making components *parametrizable* over different aspects affecting their behaviour. *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].

While McIlroy's vision was warmly received, the truth is that to date that vision has not been fully realised, largely due to limitations of current programming languages. For example, the majority of languages have a bias towards one kind of decomposition of software systems, which imposes a corresponding bias on the kinds of *extensibility* available: some languages favour *object-oriented decomposition* (where adding new variants is easy), while others favour *functional decomposition* (where adding new functions is easy). Extensibility is important for the development of components [Szyperski, 1996], yet the bias imposed by a language makes it hard to develop components that require the dual kind of extensibility. A related problem is 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], the structure of which cannot be expressed more precisely than in terms of prose, pictures and prototypes.

Our first contribution in this paper 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 structural 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?*' [Gamma et al., 1995] — the operation or the object structure itself?
- is the visitor *imperative* (with results of traversals stored as mutable state in the visitor) or *functional* (with results returned by the *accept* method)?
- does the visitor satisfy certain *orthogonal concerns* such as tracing or caching of computations?

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 modular 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* [Cremet and Altherr, 2008] could be used instead). As far as we are aware, all existing solutions in the literature trying to capture some notion of generic visitors [Palsberg and Jay, 1998, Visser, 2001, Orleans and Lieberherr, 2001, Grothoff, 2003, Forax et al., 2005, Meyer and Arnout, 2006] make use of reflection, introspection or metaprogramming mechanisms that do not statically guarantee type-safety. Furthermore, most of those solutions only capture particular variants of the pattern.

Our second contribution is a semantics for a generalised *algebraic datatype notation* using visitors built with our library. The notation allows 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. We believe that this notation is practical and can significantly reduce the burden of expressing functional decomposition in an object-oriented language.

### 1.1 Overview

Section 2 introduces the VISITOR design pattern, and the problems it induces. In Section 3, we present our visitor library from a programmer's perspective, and informally introduce the datatype notation. Section 4 reviews the work of Buchlovsky and Thielecke [2005] relating the VISITOR pattern to lambda calculus encodings of datatypes, and extends this work to obtain generic encodings of datatypes that are parametric in the traversal strategy. In Section 5, we exploit these developments in a highly generic library of visitors written in Scala. In Section 6, we present a formal translation between the datatype notation and visitors defined with the visitor library. Finally, a discussion of the results and related work is presented in Section 7, and conclusions in Section 8.

## 2. The VISITOR as a Design Pattern

In this section we review the traditional presentation of the VISITOR pattern [Gamma et al., 1995], and discuss some problems it presents. We hasten to emphasize that in this paper we are only really talking about the *structural* aspects of the design pattern, and not any of the other important aspects such as the motivating 'story' or example code. Having said that, we do believe that we have captured the most important structural variations in the implementation of the 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, and thus allowing the extension of the former without changing the latter. Moreover, the VISITOR keeps related aspects of a single operation together, by defining them in a single class. Figure 1 shows the class structure of the pattern. The participants 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 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*.

In contrast to the standard object-oriented decomposition, epitomised by the COMPOSITE pattern [Gamma et al., 1995], and like a functional 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. Borrowing some terminology from Buchlovsky and Thielecke [2005], we 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,

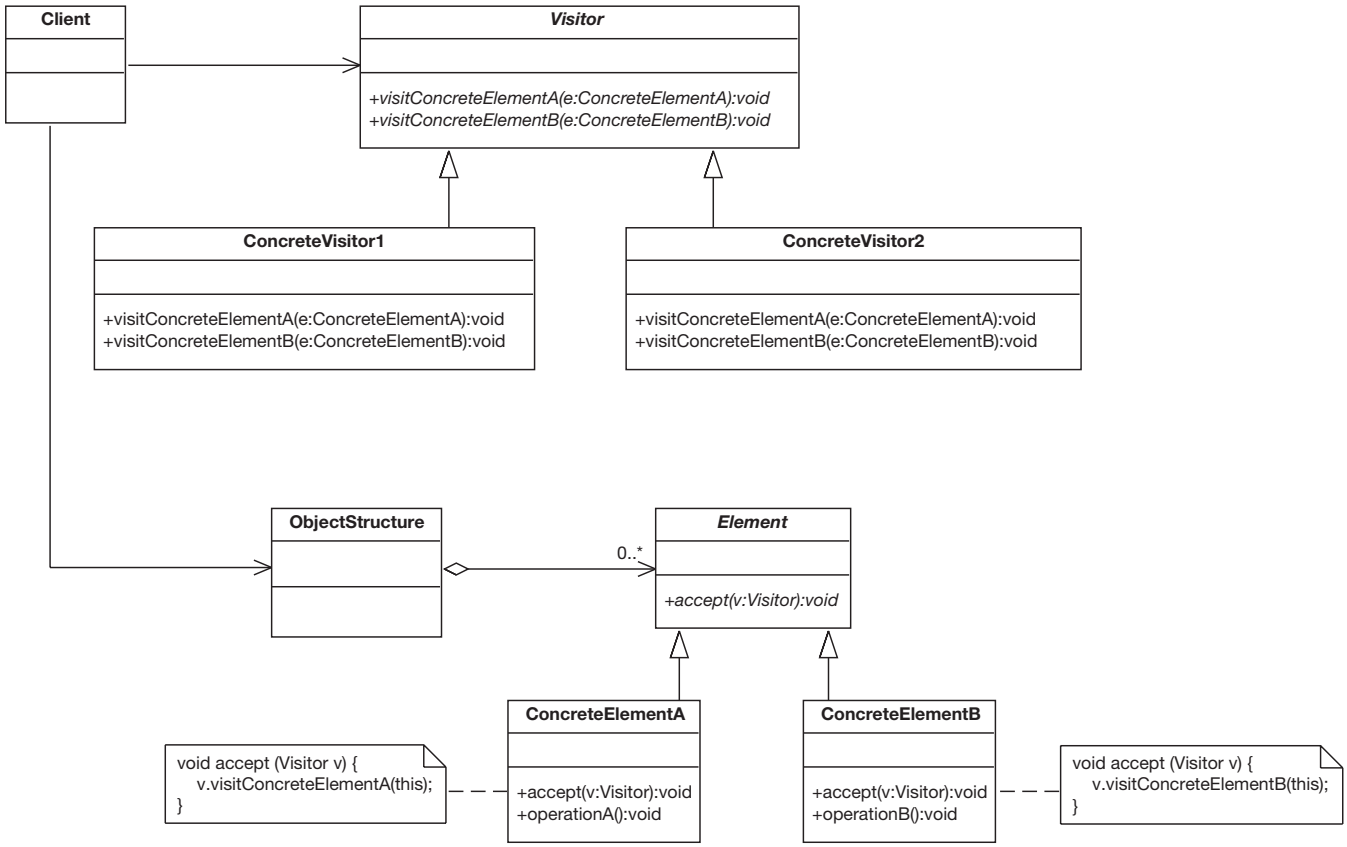


**Figure 1.** The VISITOR design pattern

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

In their presentation of the VISITOR pattern, 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 (*Empty* represents a leaf, and *Fork* a composition of two subtrees). 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. 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 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:

```
object Depth extends TreeVisitor[int] {
  def empty                          = 0
  def fork (x: int, l: Tree, 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.

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

**Internal 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
}
```

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

**Figure 2.** Internal and External VISITORs for Binary Trees

### 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 naïvely, 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

In this section we present a programmer's view of the Scala visitor library, showing how it can avoid the early design decisions imposed by the design pattern approach.

### 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:

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

An equivalent definition in our **data** notation is:

```
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.

| Library notation | VISITOR terminology |
|---|---|
| **data** $T$ | Element |
| **constructor** | Concrete Element |
| $(D)Case_T$ | Visitor |
| $V$ **extends** $(D)Case_T$ | Concrete Visitor |
| **new** $(D)Case_T$ | Anonymous Concrete Visitor |

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

***Generalized* data *notation*** We make our **data** notation more amenable to object-oriented programming by gener-

```
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 ()
}
```

**Figure 3.** Using the generalized **data** notation to define *Nat*.

alizing it so that datatypes can define and override methods and values, in the same way as classes or traits. 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.

### 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:

```
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, $depth_1$ 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, $depth_1$ can be rewritten as follows, which nicely reflects the recursive nature of the definition:

```
def depth2 = new CaseTree[External, int] {
  def Empty                  = 0
  def Fork (x: int, l: Tree, r: Tree) =
    1 + max (depth2 (l), depth2 (r))
}
```

***Internal Visitors*** In the definitions of $depth_1$ and $depth_2$ 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:

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

Since internal visitors use traversal strategies determined by the elements, the above definition does not require explicit traversal of the structure, so 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 (naïve) version of the Fibonacci function defined over *Nat*.

[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).

[2] In contrast to the pointcut mechanism in AOP, our advice is installed by parametrization. We leave a detailed comparison to Section 7.

```
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:

```
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 the library, explained in detail in Section 5). The $fib_2$ 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 the 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 $fib_2$:

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

The program *nfib* is equivalent to $fib_1$, while *mfib* is a version with memoization. The programs *tmfib* and *mtfib* combine tracing and memoization in different ways: 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 advice is determined by the order of composition. In *tmfib*, *Trace* is triggered before *Memo*, which prints out all calls including those resorting to memoization. On the other hand, *mtfib* 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 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, consider adding all the integers in some tree, 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:

```
class AddTree1 extends CaseTree[Internal, Unit] {
  var sumValue = 0
  def Empty                    = {}
  def Fork (x: int, l: Unit, r: Unit) =
    {sumValue += x;}
}
```

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

```
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 equivalent to $l$.*accept* (**this**)). The imperative visitors are used as follows:

```
def test: int = {
  val addTree = new AddTree_n ();
  val tree1   = Fork (3, Empty (), Empty ());
  val tree2   = Fork (4, tree1, tree1);
```

```
    addTree (tree2);
    return addTree.sumValue;
  }
```

Here, $AddTree_n$ should be replaced by either $AddTree_1$ or $AddTree_2$. The program creates a new instance *addTree* of $AddTree_n$, 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 modularly type-safe (in the sense that no casts and no global analysis 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:

```
equal :: Tree → Tree → Bool
equal Empty          Empty          = True
equal (Fork x l1 r1) (Fork y l2 r2) =
  x ≡ y ∧ equal l1 l2 ∧ equal r1 r2
equal _              _              = False
```

Support for pattern-matching could be built on top of external visitors in essentially the same way that it is built on top of case analysis in most functional programming languages; we leave the details of such an extension for future work.

### 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 Tree[a] {
  constructor Fork (x:a, f:Forest[a])
}
data Forest[a] {
  constructor Nil
  constructor Cons (t:Tree[a], f:Forest[a])
}
```

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

We could define a function to sum all the leaves of a tree of integers as follows:

```
def sumTree = new CaseTree[Internal, int, int] {
  def mrefForest = sumForest
  def fork (x:int, xs:int) = x + xs
}
def sumForest = new CaseForest[Internal, int, int] {
  def mrefTree = sumTree
  def nil          = 0
  def cons (x:int, xs: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 visitors contain fields referring to the visitors that they depend on. We name such fields *mrefForest* and *mrefTree* (the details are explained in Section 6). Additionally, for parametrized types like $Tree[a]$, type arguments (such as *a*) are also passed as arguments to *CaseTree*.

```
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, l1:Tree, r1: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.

**Church Encodings**

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

**Parigot Encodings**

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

**Figure 5.** Encodings of naturals and binary trees.

## 4. Visitors as Encodings of Datatypes

In this section, we look at the relationship between visitors and encodings of datatypes, and introduce the theoretical foundations for the Scala visitor library presented in Section 5.

### 4.1 Encoding Datatypes in the Lambda Calculus

The pure lambda calculus has no notion of datatypes; they have to be encoded using functions. Church [1936] showed how to encode the natural numbers via 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.

$\mathit{zero} \equiv \lambda f \Rightarrow \lambda x \Rightarrow x$
$\mathit{succ} \equiv \lambda n \Rightarrow \lambda f \Rightarrow \lambda x \Rightarrow f\ (n\ f\ x)$

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 Beraducci'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. Spławski 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 *Nat* and *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 *Nat* and *Tree* need to define the traversal themselves. This requires that the types of those functions recursively refer to *Tree* and *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 *Nat* instead of *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 knowledge 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:

$T \equiv \forall X.\ (F\ R \Rightarrow X) \Rightarrow X$

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

$T \equiv \forall X.\ ((\Sigma_i\ F_i\ R) \Rightarrow X) \Rightarrow X$

Now, the type $(\Sigma_i\ F_i\ R) \Rightarrow X$ of functions from a sum is isomorphic to the type $\Pi_i\ (F_i\ R \Rightarrow X)$ of products of functions (in the same way that $x^{y+z} = x^y \times x^z$); 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, respectively, to internal and external visitors — follow from two

$$
\begin{array}{ll}
\mathit{NatF}\ R\ A \equiv (A, R \Rightarrow A) \\
\mathit{Nat} & \equiv \mathit{Internal}\ \mathit{NatF} \\
\mathit{zero} & \in \mathit{Nat} \\
\mathit{zero} & \equiv \lambda(z,s) \Rightarrow z \\
\mathit{succ} & \in \mathit{Nat} \Rightarrow \mathit{Nat} \\
\mathit{succ}\ n & \equiv \lambda(z,s) \Rightarrow s\ (n\ (z,s))
\end{array}
$$

**Figure 6.** Church encoding of Peano numerals using products of functions

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

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

- *Generic external visitors* are obtained by specializing $R \equiv \mathit{External}\ V$; we can define

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

In each case, $V$ is a type parameter abstracting over concrete visitor components. It could be said that $V$ is the *shape parameter* of the encodings, since different instantiations of $V$ will lead to different datatypes.

### 4.3 Generic Visitors: Traversal Strategy Abstraction

Generic encodings based on products of functions allow one to abstract from differences in the shape of data and model different traversal strategies — internal and external — of datatype-generic visitors. Still, there is substantial duplication of code whenever we want to have both strategies. However, this duplication can be avoided: we can model visitors that are generic in both the shape and the traversal strategy. The template

$\mathit{Composite}\ V \equiv \forall X.\ V\ R\ X \Rightarrow X$

could be used to capture different implementations of the VISITOR pattern by using a proper instantiation for $R$. However, this definition is not valid in System F, because $R$ is unbound; some other approach is needed. Since $R$ represents the type of recursive occurrences that appear in the visit methods, if we want to capture both internal and external visitors, $R$ should depend on both $V$ and $X$. This dependency can be made explicit by having $R \equiv S\ V\ X$ and binding $S$ universally.

$\mathit{Composite}\ V \equiv \forall S\ X.\ V\ (S\ V\ X)\ X \Rightarrow X$

We shall refer to $S$ as the *traversal strategy*.

Although *Composite V* is now a valid System F definition, 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*, there are no problems in defining the constructor *zero*; however for *succ*, it is impossible to provide a value of the right type:

$\mathit{Nat} \equiv \mathit{Composite}\ \mathit{NatF}$

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

$s$ requires an argument with type $S\ V\ X$, and we cannot create any values of that type. The solution for this problem consists in adding some extra information about $S$ in the definition of *Composite*.

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

The extra information is given by *Decompose S*, which 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, made implicit. Referring to the method in *Decompose S* as $dec_S$, we have that:

$dec_S \in V\ (S\ V\ X)\ X \Rightarrow \mathit{Composite}\ V \Rightarrow S\ V\ X$

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

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

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 $dec_S$ operation. For example, to make internal and external visitors two instances of *Composite V*, we specialize $S$ to *Internal* and *External*:

$\mathit{Internal}\ V\ X\ \equiv X$
$\mathit{External}\ V\ X \equiv \mathit{Composite}\ V$

Here we reuse the identifiers *Internal* and *External* to refer to the associated traversal strategies. The specific instantiations of $dec_S$ for internal and external visitors are:

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

In the definition of $dec_{\mathit{Internal}}$ the reader should (again) note that the *Decompose S* parameter is implicitly passed and, therefore, the composite $c$ just needs to take the visitor $v$ as an argument. With $dec_{\mathit{External}}$, 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 *Composite*, and annotate it with extra information identifying the *accept* method and the visitor component.

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

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 [2005b] to be essential for the construction of reusable components; they allow information hiding over several objects, a key part of component-oriented programming [Pfister and Szyperski, 1996].

Alternatively to abstract types, we could have used *type-constructor polymorphism* [Cremet and Altherr, 2008] 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).

```
trait Visitor {
  type X
  type S <: Strategy
  type R[v <: Visitor] =
    (S {type X = Visitor.this.X; 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 *VisFunc* parametrized by a visitor *v*, a strategy *s* and a result type *x*, as a shortcut for visitors that are also functions.

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

In essence, we treat visitors as functions (the trait *Function1* is provided by the Scala library) that take a *Composite*$[v]$ as an argument and return a value of type *x*, by observing that the invocation $a.\mathit{accept}\ (f)$ 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 *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.

```
trait Composite[v <: Visitor] {
  def accept[s <: Strategy, x] (vis: VisFunc[v, s, x])
      (implicit decompose: 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.

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

```
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:

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

```
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
}
```

Traversal 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 : 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. Note that the *apply* method is defined in the *Function1* trait and corresponds to function application. 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 (that is, we can construct more complex dispatchers using simpler ones) 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*:

```
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} ⇒
    type X = x
    type S = s
    def dispatcher = d
    def decompose = dec
    def apply (c : Composite[v]) : x =
      dispatcher.dispatch (this, decompose).apply (c)
  }
```

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* = *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

```
trait Dispatcher[v<:Visitor, s<:Strategy, x] {
  def dispatch(vis: VisFunc[v,s,x], dec: Decompose[s]): Function1[Composite[v],x]
}
implicit def Basic[v<:Visitor, s<:Strategy, x] = new Dispatcher[v,s,x] {
  def dispatch(vis: VisFunc[v,s,x], dec: Decompose[s]): Function1[Composite[v],x] =
    c => c.accept[s,x](vis)(dec)
}
abstract class Advice[d<:Visitor, s<:Strategy, x](dis: Dispatcher[d,s,x]) extends Dispatcher[d,s,x] {
  def before(comp: Composite[d]): Unit = {}
  def after(comp: Composite[d], res: x): Unit = {}
  def dispatch(vis: VisFunc[d,s,x], dec: Decompose[s]): Function1[Composite[d],x] =
    c => {before(c); val res = dis.dispatch(vis,dec)(c); after(c,res); res}
}
def Trace[v<:Visitor, s<:Strategy, x](dis: Dispatcher[v,s,x]) = new Advice[v,s,x](dis) {
  override def before(comp: Composite[v]): Unit = {
    System.out.println("Calling function with argument: \t" + comp);
  }
  override def after(comp: Composite[v], res: x): Unit = {
    System.out.println(res + "\t was returned from the call with argument: \t" + comp);
  }
}
def Memo[v<:Visitor, s<:Strategy, x](dis: Dispatcher[v,s,x]) = new Dispatcher[v,s,x] {
  val cache: HashMap[Composite[v],x] = new HashMap[Composite[v],x]()
  def dispatch(vis: VisFunc[v,s,x], dec: Decompose[s]): Function1[Composite[v],x] = c => {
    cache.get(c) match {
      case Some(x) => x
      case None => {val res = dis.dispatch(vis,dec)(c); cache.put(c,res); res}
      }
  }
}
```

**Figure 7.** Visitor Library Dispatchers

as follows.

| | | | |
|---|---|---|---|
| Datas | $\tau$ | ::= | **data** $\mathcal{T}$ $[\overline{\alpha}] = \{\overline{c}\ \overline{s}\}$ |
| Constructors | $c$ | ::= | **constructor** $\mathcal{K}$ $[\beta]$ $\overline{v:t}$ $\{\overline{s}\}$ |
| Types | $t$ | ::= | $t_1 \mid \mathcal{T}_0\ [\overline{\alpha}]$ |
| Non-recursive Types | $t_1$ | ::= | $\alpha \mid \mathcal{T}_1\ [\overline{t_1}] \mid t_1 \to t_1$ |
| Scala | $s$ | ::= | Scala declarations |

A datatype $\mathcal{T}$, possibly parametrized by type variables $\overline{\alpha}$, introduces a set of data constructors and some optional Scala code $\overline{s}$. Each constructor, $\mathcal{K}$, can take an optional list of type arguments $\beta$ (which act as existentially quantified types) and a list of labelled type arguments $\overline{v:t}$. Scala definitions $\overline{s}$ can be inserted to define or override fields and methods. We single out non-recursive type arguments, $t_1$, which do not make self-reference to the datatype that introduced them. Recursive occurrences of type constructors are denoted $\mathcal{T}_0$, to separate them from the non-recursive ones ($\mathcal{T}_1$).

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 **data** constructor presented here is comparable in expressive power to ML-style and Haskell 98-style datatypes, allowing us to express *(type-)parametrized datatypes*, *mutually recursive datatypes* and *existential datatypes*.

Declarations in the datatype language can be translated to visitors by the meta-function GEN in Figure 8. Before going into the details of the translation, we first introduce a few notational conventions. We write $\overline{o}^n$ for a sequence of entities numbered from 1 to $n$ and $o_i$ as the $i^{th}$ of them. We use a pattern matching syntax $t_i@\mathcal{T}\ [\overline{\gamma}]$ to denote that the bound variable $t$ is of type $\mathcal{T}\ [\overline{\gamma}]$ for some $\mathcal{T}$ and $\gamma$. New

```
GEN(data T [ᾱ] = {c̄ s̄}) =
  LET
    GENREF(T_m) = def mref T_m : VisFunc[T_m Visitor[ᾱ], S, X]
    GENTYPE(constructor K [β̄] v̄ : t̄^n {s̄}) =
      def K [β̄] (v_i : CASE STATUSOF(t_i@T'[γ̄]) OF RECURSIVE  → R[T Visitor[ᾱ]]
                                          MUTUALREC → R[T' Visitor[ᾱ]]
                                          NONREC    → t_i)^{i∈1..n} : X
    GENDATA(constructor K [β̄] v̄ : t̄^n {s̄}) =
      case class K [β̄, ᾱ] (v_i : t_i)^{i∈1..n} extends T [ᾱ] {
        def accept[s <: Strategy, x] (vis : VisFunc[T Visitor[ᾱ], s, x]) (implicit decompose : Decompose[s]) : x =
          vis.K [β̄] (CASE STATUSOF(t_i@T'[γ̄]) OF RECURSIVE  → decompose.dec[T Visitor[ᾱ], x] (vis, v_i)
                                          MUTUALREC → decompose.dec[T' Visitor[ᾱ], x] (vis.mref T', v_i)
                                          NONREC    → v_i)^{i∈1..n}
        s̄
      }
  IN
    trait T Visitor[ᾱ] extends Visitor {
      GENREF(T_m)
      GENTYPE(c̄)
    }
    trait T [ᾱ] extends Composite[T Visitor[ᾱ]]{s̄}
    abstract class CaseT [s <: Strategy, ᾱ, x] (implicit dec : Decompose[s]) extends DCaseT [s, ᾱ, x] (Basic) (dec)
    abstract class DCaseT [s <: Strategy, ᾱ, x] (disp : Dispatcher[T Visitor[ᾱ], s, x]) (implicit dec : Decompose[s])
      extends Case[T Visitor[ᾱ], s, x] (disp) (dec) with T Visitor[ᾱ]
    GENDATA(c̄)
```

**Figure 8.** Translation Scheme

names for visitors and references are created by prefixing or postfixing with the type constructor name, for example $\mathcal{T}$*Visitor*. We assume a dependency analysis and write $\mathcal{T}_m$ to denote the set of mutually recursive types that $\mathcal{T}$ makes references to (excluding $\mathcal{T}$ itself).

For each datatype $\mathcal{T}$, we generate a corresponding visitor type (a trait that extends *Visitor*) and a composite (a trait that extends *Composite*$[\mathcal{T}$*Visitor*$[\bar{\alpha}]]$). We also generate two auxiliary visitors *DCase*$\mathcal{T}$ and *Case*$\mathcal{T}$. The former extends *Case*$[\mathcal{T}$*Visitor*$[\bar{\alpha}], s, x]$, 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 GENDATA creates a case class for each constructor $\mathcal{K}$ extending $\mathcal{T}$ $[\bar{\alpha}]$ and generates the corresponding *accept* method by checking the recursive status of $\mathcal{K}$'s arguments, which determines the traversal code.

Each of the visitors $\mathcal{T}$*Visitor*$[\bar{\alpha}]$ may have mutually recursive references to other visitors that it depends on, which are generated by the function GENREF. The types of the *visit* methods $\mathcal{K}$ (named after the corresponding constructor) also depend on the recursive status of the constructor's arguments and are generated by the GENTYPE function.

In Figure 9 we apply the translation to the trees and forests example in Section 3.6. For the datatype *Tree*$[a]$, we generate the visitor and composite types *TreeVisitor*$[a]$ and *Tree*$[a]$, the two auxiliary visitors *CaseTree* and *DCaseTree*, and the constructor *Fork*. The mutual dependency with *Forest*$[a]$ is captured by the definition of *mrefForest* in *TreeVisitor*$[a]$. A similar process happens for *Forest*$[a]$, resulting in the generation of *ForestVisitor*$[a]$, *Forest*$[a]$, *CaseForest*, *DCaseForest*, *Nil* and *Cons*. A mutual reference *mrefTree* is also placed in *ForestVisitor*$[a]$.

## 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 Moor, 1997]. This line of 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

```
trait TreeVisitor[a] extends Visitor {
  def mrefForest: VisFunc[ForestVisitor[a], S, X]
  def Fork (x: a, xs: R[ForestVisitor[a]]): X
}
trait Tree[a] extends Composite[TreeVisitor[a]]
abstract class CaseTree[s <: Strategy, a, x] (implicit dec: Decompose[s]) extends DCaseTree[s, a, x] (Basic) (dec)
abstract class DCaseTree[s <: Strategy, a, x] (disp: Dispatcher[TreeVisitor[a], s, x]) (implicit dec: Decompose[s])
  extends Case[TreeVisitor[a], s, x] (disp) (dec) with TreeVisitor[a]
case class Fork[a] (x: a, xs: Forest[a]) extends Tree[a] {
  def accept[s <: Strategy, x] (vis: VisFunc[TreeVisitor[a], s, x]) (implicit decompose: Decompose[s]): x =
    vis.Fork (x, decompose.dec (vis.mrefForest, xs))
}
trait ForestVisitor[a] extends Visitor {
  def mrefTree: VisFunc[TreeVisitor[a], S, X]
  def Nil: X
  def Cons (x: R[TreeVisitor[a]], xs: R[ForestVisitor[a]]): X
}
trait Forest[a] extends Composite[ForestVisitor[a]]
abstract class CaseForest[s <: Strategy, a, x] (implicit dec: Decompose[s]) extends DCaseForest[s, a, x] (Basic) (dec)
abstract class DCaseForest[s <: Strategy, a, x] (disp: Dispatcher[ForestVisitor[a], s, x]) (implicit dec: Decompose[s])
  extends Case[ForestVisitor[a], s, x] (disp) (dec) with ForestVisitor[a]
case class Nil[a] extends Forest[a] {
  def accept[s <: Strategy, x] (vis: VisFunc[ForestVisitor[a], s, x]) (implicit decompose: Decompose[s]): x =
    vis.Nil
}
case class Cons[a] (x: Tree[a], xs: Forest[a]) extends Forest[a] {
  def accept[s <: Strategy, x] (vis: VisFunc[ForestVisitor[a], s, x]) (implicit decompose: Decompose[s]): x =
    vis.Cons (decompose.dec (vis.mrefTree, x), decompose.dec (vis, xs))
}
```

**Figure 9.** Translation of the *Tree* and *Forest* datatypes into visitors.

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:

```
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)
}
```

### 7.2 Dispatching Strategies and Modular Concerns

Kiczales et al. [1997]'s aspect-oriented programming (AOP) aims at modularizing concerns that cut across the compo-

nents 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* from functional programming. Scala [Odersky, 2006] has its own notion of algebraic datatypes via (sealed) case classes. With case classes, we could rewrite the *Tree* and *depth* examples as:

```
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 algebraic datatypes. However, simple case classes are more general than algebraic datatypes, because they do not need to be sealed: we could have defined *Tree* without the **sealed** keyword, gaining the ability to add 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*" run-time errors.

There are three main differences between the notion of datatypes introduced in this paper and case classes. Firstly, algebraic datatypes 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 extend the functionally provided by 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 Design Patterns as Software Components

Norvig [1996] studied the consequences of using a dynamic language such as Lisp or Dylan for the GoF patterns [Gamma et al., 1995]; he claimed that 16 of these 23 patterns have qualitatively simpler implementations in such languages — some simply disappear, and others may be formalized as software components. Arnout [2004] did a similar study of design patterns in the Eiffel programming language [Meyer, 1997]; she argued that Eiffel features such as *genericity*, *tuples* and *agents* played an essential role in the componentization of patterns. Hannemann and Kiczales [2002] studied design patterns in the context of AspectJ. The results showed that 74% of the GoF patterns could be implemented in a more modular way and 52% were more reusable. Gibbons [2003] argues that datatype-generic programming can be used to capture the abstractions behind many design patterns formally; subsequent papers [Gibbons, 2006, Gibbons and Oliveira, 2008] interpret four of the GoF patterns as higher-order datatype-generic programs. Odersky and Zenger [2005b] point out that the Scala programming language is designed with component development in mind; 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, and use the first two features to provide an elegant software component that captures the OBSERVER design pattern.

### 7.5 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.

The *Peripaton* language [VanDrunen and Palsberg, 2004] supports the so-called "*visitor-oriented programming style*". 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, similarly to the functional notation in our visitor library. DJ [Orleans and Lieberherr, 2001] is a reflection-based Java library for traversals that has been used in the context of *adaptive programming*. Chadwick et al. [2006] proposes a (functional) visitor-oriented programming language that is modular and compositional, aiming at more reusable designs. They have implemented the idea as a modified version of DJ. In subsequent work, Chadwick and Lieberherr [2008] propose a reflection-based Java library DemeterF that comes with a type system making it possible to type-check visitor code, verifying traversals statically.

### 7.6 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. Multiple dispatching makes it difficult to provide modular (compile-time) type-checking to catch ambiguous and invalid combinations of dynamically dispatched arguments, and there are fears that it goes against object-oriented principles like encapsulation. 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 achieve modular static type-checking without restrictions.

Visitors can be used to emulate a (limited) form of multiple dispatching in object-oriented languages, as we mentioned in Section 3.5. 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. The problem of extensibility of visitors (that is, the ability to add new variants) is explored by Oliveira [2007, Chapter 5], and a solution inspired by Odersky and Zenger [2005a] is proposed as a way to add extensibility to the visitor library. 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, in some situations 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 structural aspects of) the VISITOR design pattern can be captured as a reusable, generic, modular and statically type-safe component by using some advanced type system features that are starting to appear in modern object-oriented languages. Inspired by functional programming, we have shown that we can significantly improve the use of visitors via datatype-like and functional no-

tations, 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 [Oliveira, 2007, Chapter 3], the first author's doctoral dissertation; in essence, this 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 these two problems are also achieved without compromising modular 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 language 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


Jonathan Aldrich. Open modules: Modular reasoning about advice. In *LNCS 3586: European Conference on Object-Oriented Programming*, pages 144–168, 2005.

Karine Arnout. *Pattern Componentization*. PhD thesis, Swiss Institute of Technology, March 2004.

Richard Bird and Lambert Meertens. Nested datatypes. In J. Jeuring, editor, *LNCS 1422: Mathematics of Program Construction*, pages 52–67, 1998.

Richard S. Bird and Oege De Moor. *Algebra of Programming*. Prentice Hall, 1997.

Corrado Böhm and Alesssandro Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theoretical Computer Science*, 39(2-3):135–153, August 1985.

Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the Visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155, 2005. Mathematical Foundations of Programming Semantics.

Bryan Chadwick and Karl Lieberherr. Functional Adaptive Programming with DemeterF. Technical Report NU-CCIS-08-March19, Northeastern University, Boston, March 2008.

Bryan Chadwick, Therapon Skotiniotis, and Karl Lieberherr. Functional Visitors Revisited. Technical Report NU-CCIS-06-03, Northeastern University, Boston, May 2006.

Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, 2000.

Vincent Cremet and Philippe Altherr. Adding type constructor parameterization to Java. *Journal of Object Technology*, 7(5):25–65, June 2008. Special issue on ECOOP 2007 Workshop on Formal Techniques for Java-like Programs.

Antonio Cunei and Jan Vitek. PolyD: a flexible dispatching framework. In *Object Oriented Programming, Systems, Languages, and Applications*, pages 487–503, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1103845.1094849.

Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object-Oriented Programming*, pages 186–211, London, UK, 1998. Springer-Verlag.

Rémi Forax, Etienne Duris, and Gilles Roussel. Reflection-based implementation of Java extensions: The double-dispatch use-case. In *ACM Symposium on Applied Computing*, pages 1409–1413, 2005.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Jeremy Gibbons. Patterns in datatype-generic programming. In J. Striegnitz, editor, *Declarative Programming in the Context of Object-Oriented Languages, Uppsala*, 2003.

Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In *ACM Workshop on Generic Programming*, pages 1–12, 2006.

Jeremy Gibbons and Bruno Oliveira. The essence of the Iterator pattern. *Journal of Functional Programming*, 2008.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.

Christian Grothoff. Walkabout revisited: The Runabout. In *LNCS 2743: European Conference on Object-Oriented Programming*, pages 103–125, 2003.

Stephan Gudmundson and Gregor Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *ECOOP 2001 Workshop on Advanced Separation of Concerns*, 2001.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11): 161–173, 2002.

Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Principles of Programming Languages*, pages 123–137, 1994.

Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ACM: International Conference on Software engineering*, pages 49–58, 2005.

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *LNCS 1241: European Conference on Object-Oriented Programming*, pages 220–242, 1997.

Xavier Leroy. Manifest types, modules, and separate compilation. In *Principles of Programming Languages*, pages 109–122, 1994.

Doug McIlroy. Mass produced software components. In Naur and Randell [1969], pages 138–155.

Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–425, 1992.

Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Functional Programming Languages and Computer Architecture*, pages 324–333, 1995.

Bertrand Meyer. *Object-Oriented Software Construction*. Upper Saddle River, N.J., Prentice Hall, 2nd edition, 1997.

Bertrand Meyer and Karine Arnout. Componentization: The Visitor example. *Computer*, 39(7):23–30, 2006.

Donald Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2007.

Peter Naur and Brian Randell, editors. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968*. 1969.

Peter Norvig. Design patterns in dynamic programming. In *Object World 96*, May 1996.

Martin Odersky. An Overview of the Scala programming language (second edition). Technical Report IC/2006/001, EPFL Lausanne, Switzerland, 2006.

Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proceedings of Foundations of Object-Oriented Languages 12*, January 2005a. `http://homepages.inf.ed.ac.uk/wadler/fool`.

Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Object Oriented Programming, Systems, Languages, and Applications*, pages 41–57, 2005b.

Bruno C.d.S. Oliveira. *Genericity, Extensibility and Type-Safety in the* VISITOR *Pattern*. PhD thesis, University of Oxford, 2007.

Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag.

Jens Palsberg and C. Barry Jay. The essence of the Visitor pattern. In *Computer Software and Applications*, pages 9–15, 1998.

Michel Parigot. Recursive programming with proofs. *Theoretical Computer Science*, 94(2):335–356, 1992.

Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, Cambridge, England, 2003.

Cuno Pfister and Clemens Szyperski. Why objects are not enough. In *International Component Users Conference*, 1996.

Zdzisław Spławski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. In *International Conference on Functional Programming*, pages 102–113, 1999.

Clemens Szyperski. Independently extensible systems – software engineering potential and challenges. In *19th Australian Computer Science Conference*, 1996.

Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.

Thomas VanDrunen and Jens Palsberg. Visitor-oriented programming. In *Proceedings of FOOL-11, the 11th ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, New York, NY, USA, January 2004. ACM Press.

Joost Visser. Visitor combination and traversal control. In *Object Oriented Programming, Systems, Languages, and Applications*, pages 270–282. ACM, 2001.