Synthesizing Object-Oriented and Functional Design to Promote Re-Use*

Shriram Krishnamurthi Matthias Felleisen
Daniel P. Friedman†

Department of Computer Science
Rice University

April 9, 1998

A preliminary version of this paper appeared in the
European Conference on Object-Oriented Programming, 1998

Abstract

Many problems require recursively specified types of data and a collection of tools that operate on those data. Over time, these problems evolve so that the programmer must extend the toolkit or extend the types and adjust the existing tools accordingly. Ideally, this should be done without modifying existing code. Unfortunately, the prevailing program design strategies do not support both forms of extensibility: functional programming accommodates the addition of tools, while object-oriented programming supports either adding new tools or extending the data set, but not both. In this paper, we present a composite design pattern that synthesizes the best of both approaches and in the process resolves the tension between the two design strategies. We also show how this protocol suggests a new set of linguistic facilities for languages that support class systems.

1 Evolutionary Software Development

Programming practice frequently confronts programmers with the following design dilemma. A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. add a new variant of data and adjust the existing tools accordingly, and

2. extend the collection of tools.

Ideally, these extensions should not require any changes to existing code. For one, source modification is cumbersome and error-prone. Second, the source may not be available for modification because the tools are distributed in the form of object code. Finally, it may be necessary to evolve the base

*This research was partially supported by NSF grants CCR-9619756, CCR-9633109, CCR-9708957 and CDA-9713032, and a Texas ATP grant.

†Permanent address: Computer Science Department, Indiana University.

program in several different directions, in which case code modifications are prohibitively expensive because the required duplication would result in duplicated maintenance costs.

This dilemma manifests itself in many different application areas. A particularly important example arises in the area of programming languages. Language grammars are typically specified via BNFs, which denote recursively defined data sets. Language-processing tools recursively traverse sentences formed from the grammar. In this scenario, a new form of data means an additional clause in the BNF; new tools must be able to traverse all possible elements of the (extended) grammar.

Unfortunately, prevailing design strategies do not accommodate the required evolution:

- The "functional" approach, which is often realized with conventional procedural languages, implements tools as procedures on recursive types. While this strategy easily accommodates the extension of the set of tools, it requires significant source modifications when the data set needs to be extended.
- The (standard) "object-oriented" approach defines a recursive set of data with a collection of classes, one per variant (BNF clause), and places one method per tool in each class. In the parlance of object-oriented design patterns [13], this approach is known as the Interpreter pattern. The problem it poses is dual to the problem of the functional approach: variants are easy to add, while tool additions require code modifications.
- If the collection of tools is large, the designer may also use the Visitor pattern, a variant of the Interpreter pattern, which collects the code for a tool in a single class. Roughly speaking, the Visitor pattern emulates the functional approach in an object-oriented setting. As a result, it suffers from the same problem as the functional approach.

In short, the two design styles suffer from a serious problem. Each style accommodates one form of extension easily and renders the other nearly impossible.[1]

This paper presents the Extensible Visitor pattern, a new composite design pattern [28], which provides an elegant solution to the above dilemma. The composite pattern is a combination of the Visitor and Factory Method patterns. Its implementation in any class-based object-oriented programming language is straightforward. In addition, the paper introduces a linguistic abstraction that facilitates the implementation of the Visitor and Extensible Visitor patterns. The abstraction syntactically synthesizes the best of the functional and the object-oriented design approaches. Using the abstraction, a programmer only specifies the necessary pieces of the pattern; a translator assembles the pattern implementation from these pieces. We consider this approach a promising avenue for future research on pattern implementations.

Section 2 introduces a simple example of the design dilemma and briefly discusses the functional approach and the standard object-oriented approach (based on the Interpreter pattern) to extensible software. Section 3 analyzes the problems of the Visitor pattern and then develops the Extensible Visitor pattern in the context of the same running example. Section 4 describes some of the type-checking issues that arise when using this pattern. Section 5 presents a linguistic extension that facilitates the implementation of the Visitor and Extensible Visitor patterns. Section 6 discusses the state of our implementation and our experiences. The last two sections describe related work and summarize the ideas in this paper.

2 Existing Design Approaches

To illustrate the design problem with a concrete example, we present a simplistic "geometry manager" program, derived from a US Department of Defense programming contest [15]. We discuss both the functional and the object-oriented design methods in this context and expose their failings.

[1] Cook [4] devotes his tutorial to this problem, which was first anticipated by Reynolds [27].

$$
\begin{aligned}
\textbf{datatype}\ \mathsf{Shape} = {} & \square\ \textbf{of}\ \mathsf{num} \\
\mid {} & \bigcirc\ \textbf{of}\ \mathsf{num} \\
\mid {} & \cdot \rightsquigarrow \cdot\ \textbf{of}\ \mathsf{Point} \times \mathsf{Shape}
\end{aligned}
$$

Figure 1: The Functional Approach: Types

$$\mathsf{ContainsPt} : \mathsf{Point} \times \mathsf{Shape} \longrightarrow \textbf{boolean}$$

$$
\begin{aligned}
\mathsf{ContainsPt}\ & p\ (\square\ s) = \cdots \\
\mid\ & p\ (\bigcirc\ r) = \cdots \\
\mid\ & p\ (\cdot \rightsquigarrow \cdot\ d\ s) = \cdots\ \mathsf{ContainsPt}\ p'\ s\ \cdots
\end{aligned}
$$

Figure 2: The Functional Approach: Tools

$$\mathsf{Shrink} : \mathsf{num} \times \mathsf{Shape} \longrightarrow \mathsf{Shape}$$

$$
\begin{aligned}
\mathsf{Shrink}\ & pct\ (\square\ s) = (\square \cdots) \\
\mid\ & pct\ (\bigcirc\ r) = (\bigcirc \cdots) \\
\mid\ & pct\ (\cdot \rightsquigarrow \cdot\ d\ s) = (\cdot \rightsquigarrow \cdot\ d\ (\mathsf{Shrink}\ pct\ s))
\end{aligned}
$$

Figure 3: The Functional Approach: Adding Tools

For this discussion, we use the term *tool* to refer to a service provided by the program, which is typically implemented as a class, function, or procedure.

Initially, our system specifies a set of data (Shape) partitioned into three subsets—squares ($\square$), circles ($\bigcirc$) and translated shapes ($\cdot \rightsquigarrow \cdot$)—and a tool that, given a shape and a point, determines whether the point is inside the shape (ContainsPt). The set of shapes is then extended with a composite shape that is the union of two others (⧉). The set of tools grows to include a shrinker that, given a number and a shape, creates a copy of that shape shrunken in its dimensions by the given percentage (Shrink).

2.1 The Functional Approach

In a functional language, recursively defined data are specified using *datatype* declarations. Such a declaration introduces a new type with one or more *variants*. In Haskell [16] or SML [20], for example, a programmer could use the **data** or **datatype** construct, respectively, to represent the set of shapes, as shown in Fig. 1.[2] Each variant introduces a new tag to distinguish it from the other forms of data. Each variant also specifies a record-like structure with a fixed number of typed fields. The types may include the datatype being declared. In the figure, the three variants of the datatype describe the structure of the different shapes: the square is described by the length of its side (a number), a circle by its radius (a number), and a translated shape by a displacement (a Point) for the underlying shape. Values are constructed by writing the name of a variant followed by as many expressions as there are fields for that variant. For example, ($\square$ 3) constructs a square, which is of type Shape and whose side has length 3.

Tools map variants of the Shape datatype to results. For example, Fig. 2 shows the outline of the tool ContainsPt, which determines whether a point is inside a shape. Its mathematics has been elided since it is rudimentary and not relevant to our example. The function definition uses pattern-matching: if a pattern matches, the identifiers to the left of = are bound on the right to the corresponding values of the fields. For example, the pattern ($\square$ s) in the first line of the function matches only squares and binds s to the length of the square's side.

Since the datatype definition of a shape is recursive, the corresponding tools are usually recursive,

[2] In C [17], one would use (recursive) pointers, structures and unions to represent this set of constructs.

too. The recursive calls in a tool match the recursive structure of the datatype. This template can be used to define other tools; for example, Fig. 3 shows the structure of Shrink, which takes a shrink factor (a number) and a shape, and produces the same shape but with the dimensions shrunk by the specified factor. We can add tools like Shrink without making any changes to existing tools such as ContainsPt.

In the functional style, the code for all the variants is defined within the scope of a single function. This simplifies the task of comprehending the tool. It also makes it easy to define abstractions over the code for the variants.

Unfortunately, it is impossible to add a variant to Shape without modifying existing code. First, the datatype representing shapes must be modified because most existing functional languages do not offer an extensible datatype mechanism at all or do so in a restricted manner [6, 19, 20]. Second, even if extensible datatype definitions are available, the code for each tool, such as ContainsPt, must be edited to accommodate these extensions to the datatype.[3]

In summary, the conventional functional programming methodology makes it easy to add new tools, but impossible to extend the datatype without code modification.

2.2 The Object-Oriented Approach

In an object-oriented program, the data definitions for shapes and their tools are developed in parallel. Abstract classes introduce new collections of data and specify signatures for the operations that are common to all variants. Concrete classes represent the variants and provide implementations of the actual operations. This is known as the Interpreter pattern [13].[4] For instance, the SML program from Figs. 1 and 2 corresponds to the Java [14] program shown in Fig. 4. The recursive references among the collection of classes lead to corresponding recursive calls among methods, analogous to the recursion in the functional program.

In this setting, it is straightforward to extend the set of shapes. It suffices to add a new concrete class that extends Shape and whose methods specify the behavior of the existing tools for that extension. For example, Fig. 5 shows how , the union of two shapes, is added to our system. Most importantly, existing tools remain unchanged.

Unfortunately, the Interpreter pattern makes it impossible to add a new tool if existing code is to stay the same. The only option is to create, for each concrete class, an extension that defines a method for the new tool. This affects every client, *i.e.*, any code that creates instances of the concrete classes. The clients must be updated to create instances of the new, extended classes instead of the old ones so that the objects they create have methods that implement the new tool.

The affected clients can include an existing tool. For example, in Fig. 6, the *shrink* method creates concrete instances of Shape that have methods for only the *containsPt* and *shrink* tools. If a tool T that is added later invokes *shrink*, the object returned by the method will not support all tools, in particular T, unless the *shrink* method is physically updated.

In summary, object-oriented programming—as represented by the Interpreter pattern—provides the equivalent of an extensible, user-defined datatype. The Interpreter pattern solves the problem of extending the set of shapes. However, this conventional design makes it difficult or, in general, impossible to extend the collection of tools without changing existing code. Furthermore, the code for each tool is distributed over several classes, which makes it more difficult to comprehend the tool's functionality. Any abstractions between the branches of a tool must reside in Shape (unless the programming language has multiple-inheritance), even though the abstraction may not apply to most tools and hence does not belong in Shape.

[3] Sometimes, the modifications may change the semantics of the operation. In such cases, a more sophisticated protocol is necessary, such as that specified by Cartwright and Felleisen [2].

[4] The Composite pattern [13] is sometimes used instead.

```
abstract class Shape {
   Shape shrink (double pct); }
class □ extends Shape {
   double s;
   □ (double s) { this.s = s ; }
   boolean containsPt (Point p) { ··· } }
class ○ extends Shape {
   double r;
   ○ (double r) { this.r = r ; }
   boolean containsPt (Point p) { ··· } }
class ·⇝· extends Shape {
   Point d;
   Shape s;
   ·⇝· (Point d, Shape s) { this.d = d ; this.s = s ; }
   boolean containsPt (Point p) {
      return (s.containsPt (···)) ; } }
```

Figure 4: The Object-Oriented Approach: Basic Types and Tools

```
class □○ extends Shape {
   Shape lhs, rhs;
   □○ (Shape lhs, Shape rhs) { this.lhs = lhs ; this.rhs = rhs ; }
   boolean containsPt (Point p) {
      return (lhs.containsPt (p) ∨ rhs.containsPt (p)) ; } }
```

Figure 5: The Object-Oriented Approach: Adding Variants

```
class Shrink□ extends □ {
   ⋮
   Shape shrink (double pct) {
      return (new Shrink□ (···)) ; }
   ⋮ }
```

```
class Shrink○ extends ○ { ··· }

class Shrink·⇝· extends ·⇝· { ··· }

class Shrink□○ extends □○ { ··· }
```

Figure 6: The Object-Oriented Approach: Adding Tools

3 A Protocol for Extensibility and Re-Use

In any interesting system, both the (recursive) data domain and the toolkit are subject to change. Thus re-use through extensibility along both dimensions is essential.

In this section, we develop a programming protocol based on object-oriented concepts that satisfies these desiderata.[5] We present the protocol in three main stages. First we explain how to represent extensible datatypes and tools via the Visitor pattern and how the Visitor pattern suffers from the same problem as the functional design strategy. Still, the Visitor pattern can be reformulated so that a programmer can extend the data domain and the toolkit in a systematic manner. Finally, we demonstrate how the protocol can accommodate extensions across multiple tools and mutually-referential data domains.

The ideas are illustrated with fragments of code written in Pizza [21], a parametrically polymorphic extension of Java. The choice of Pizza is explained in Sect. 4. In principle, any class-based language, such as C++, Eiffel, Java, or Smalltalk, suffices.

3.1 Representing Extensible Datatypes

The representation of extensible datatypes in the Visitor pattern is identical to that in the Interpreter pattern, but each class (variant) contains only one interpretive method: *process*. This method consumes a *processor*, which is an object that contains a method corresponding to each variant in the datatype. For each variant, the *process* method dispatches on that method in the processor corresponding to that variant, and returns the result of the invoked method. Figure 7 illustrates how the datatype from Sect. 2.2 is represented according to this protocol.

Since different processors return different types of results, the *process* method has the parametrically polymorphic type $\mathsf{ShapeProcessor}\langle\alpha\rangle \longrightarrow \alpha$. That is, *process*'s argument has the parametric type $\mathsf{ShapeProcessor}\langle\alpha\rangle$, which is implemented as an interface in Pizza. The return type is α in place of a single, fixed type. In Pizza, this type is written as $\langle\alpha\rangle\ \alpha$. For our running example, the parametric interface and the outline of the tool that checks for point containment (ContainsPt) are shown in Fig. 8.

If a processor depends on parameters other than the object to be processed, it accepts these as arguments to its constructor and stores them in instance variables. Thus, to check whether a point p is in a shape s, we create an instance of the ContainsPt processor, which is of type $\mathsf{ShapeProcessor}\langle\textbf{boolean}\rangle$ and which accepts the point p as an argument: **new** ContainsPt (p). This instance of ContainsPt is passed to the shape's *process* method:

$$s.process\ (\textbf{new}\ \mathsf{ContainsPt}\ (p))$$

Similarly, recursion in a processor is implemented by invoking the *process* method of the appropriate object. If the processor's extra arguments do not change, *process* can be given **this**, *i.e.*, the current instance of the processor, as its argument; otherwise, a new instance of the processor is created. Consider the ContainsPt processor in Fig. 8. It deals with translated shapes by translating the point and checking it against the underlying shape. The underlined expression in the *forTranslated* method implements the appropriate recursive call by creating a new processor.

The Visitor pattern ensures that the code for each tool is localized in a single class and easily comprehensible, as in the functional approach. In the absence of a parametrically polymorphic type system, however, it is difficult to specify the types for the Visitor pattern. Section 4 discusses this issue in detail.

[5] A preliminary version of this protocol appears in the book by Felleisen and Friedman [9].

```
abstract class Shape {
  abstract ⟨α⟩ α process (ShapeProcessor⟨α⟩ p) ; }
class □ extends Shape {
  double s;
  □ (double s) { this.s = s ; }
  ⟨α⟩ α process (ShapeProcessor⟨α⟩ p) {
    return p.forSquare (this) ; } }
class ○ extends Shape {
  double r;
  ○ (double r) { this.r = r ; }
  ⟨α⟩ α process (ShapeProcessor⟨α⟩ p) {
    return p.forCircle (this) ; } }
class · ⇝ · extends Shape {
  Point d;
  Shape s;
  · ⇝ · (Point d, Shape s) { this.d = d ; this.s = s ; }
  ⟨α⟩ α process (ShapeProcessor⟨α⟩ p) {
    return p.forTranslated (this) ; } }
```

Figure 7: The Visitor Pattern: Types

```
interface ShapeProcessor⟨α⟩ {
  α forSquare (□ s);
  α forCircle (○ c);
  α forTranslated (· ⇝ · t) ; }

class ContainsPt implements ShapeProcessor⟨boolean⟩ {
  Point p;
  ContainsPt (Point p) { this.p = p ; }
  public boolean forSquare (□ s) { ··· }
  public boolean forCircle (○ c) { ··· }
  public boolean forTranslated (· ⇝ · t) {
    return t.s.process (new ContainsPt (···)) ; } }
```

Figure 8: The Visitor Pattern: Tools

```
class Shrink implements ShapeProcessor⟨Shape⟩ {
  double pct;
  Shrink (double pct) { this.pct = pct ; }
  public Shape forSquare (□ s) { ··· }
  public Shape forCircle (○ c) { ··· }
  public Shape forTranslated (· ⇝ · t) {
    return new · ⇝ · (t.d, t.s.process (this)) ; } }
```

Figure 9: The Visitor Pattern: Adding Tools

```
interface UnionShapeProcessor⟨α⟩ extends ShapeProcessor⟨α⟩ {
  α forUnion (□○ u) ; }

class □○ extends Shape {
  Shape s1, s2;
  □○ (Shape s1, Shape s2) { ··· }
  ⟨α⟩ α process (ShapeProcessor⟨α⟩ p) {
    return ((UnionShapeProcessor) p).forUnion (this) ; } }
```

Figure 10: Datatype Extension

```
class ContainsPtUnion extends ContainsPt
  implements UnionShapeProcessor⟨boolean⟩ {
  ContainsPtUnion (Point p) { super (p) ; }
  public boolean forUnion (□○ u) {
    return u.lhs.process (this) ∨ u.rhs.process (this) ; } }
```

Figure 11: Processor Extension

3.2 Adding Tools

Extending a program's tool collection based on the Visitor pattern is straightforward. For instance, a processor that shrinks shapes would implement the ShapeProcessor⟨Shape⟩ interface. This is outlined in Fig. 9. In this example, a translated shape is shrunk by shrinking the underlying shape; the shrink factor does not change for the translated figure. Hence, the recursive call uses the same processor (**this**, underlined in the figure).

3.3 Extending the Datatype: A False Start

Since concrete subclasses represent the variants of a datatype, extending a datatype description means adding new concrete subclasses. Each new class must contain the *process* method, which is the defining characteristic of Visitor-style datatypes. The actual processors are defined separately.

In parallel to the datatype extension, we must also define an extension of the interface for processors. The extended interface specifies one method per variant in the old datatype and one for each new variant. Of course, the *process* method in the new variants should only accept processors that implement the new interface. This requirement is expressed differently in different languages. In Pizza, for example, we use a runtime check; in languages that allow *process* to be overridden covariantly, any usage errors would be caught during type-checking.

To illustrate this idea, we add the union shape (□○) to the collection of shapes. The new concrete class and interface are shown in Fig. 10. A cast (underlined in the figure) requires the processor for □○'s to implement the extended interface, UnionShapeProcessor. The extended processors can

```
class ContainsPt implements ShapeProcessor⟨boolean⟩ {
   Point p;
   ContainsPt (Point p) { this.p = p ; }
   ContainsPt makeContainsPt (Point p) {
      return new ContainsPt (p) ; }
   public boolean forSquare (□ s) { ··· }
   public boolean forCircle (○ c) { ··· }
   public boolean forTranslated (·⇝· t) {
      return t.s.process (makeContainsPt (···)) ; } }

class ContainsPtUnion extends ContainsPt
   implements UnionShapeProcessor⟨boolean⟩ {
   ContainsPtUnion (Point p) { super (p) ; }
   ContainsPt makeContainsPt (Point p) {
      return new ContainsPtUnion (p) ; }
   public boolean forUnion (◫ u) {
      return u.lhs.process (this) ∨ u.rhs.process (this) ; } }
```

Figure 12: Extensible Visitor Processor Extension

then be defined as class extensions of the existing processors for the earlier set of shapes. These extensions implement the new interface, as shown in Fig. 11.

Unfortunately, this straightforward extension of ContainsPt is incorrect. Consider the *forTranslated* method in ContainsPt. It creates a new instance of ContainsPt to process the translated shape. The new instance checks whether the "un-translated" point is in the translated shape. Since ContainsPt does not implement a *forUnion* method, a ContainsPt processor cannot process a ◫ shape. More concretely, checking whether the shape

$$\textbf{new} \cdot\!\rightsquigarrow\!\cdot\ (p,\ \textbf{new}\ ◫\ (\textbf{new}\ \square\ (\cdots),\ \textbf{new}\ \bigcirc\ (\cdots)))$$

contains some point q causes a runtime error. Specifically, when the *forTranslated* method creates a new ContainsPt processor and when this new processor is about to process the ◫ shape, the *process* method in ◫ finds that the processor does not implement the UnionShapeProcessor interface and therefore raises a runtime error.

3.4 Extending the Datatype: The Solution

The error points out that processors in the Visitor pattern are not designed to accommodate extension of the datatype. Suppose a recursive processor P can handle the variants $v_1, \ldots, v_n$. As long as the recursive call passes **this** to the datum, it does not matter whether the object is an instance of P or of a subtype of P. If, however, P creates a new instance of P for the recursive call, the new object can only handle the variants $v_1, \ldots, v_n$. When a new variant, v_{n+1}, is added, the processor provided in the recursive call can no longer process all possible inputs.

To avoid this problem, we must refrain from making a premature commitment in the recursive step. To delay making the commitment prematurely, we must delegate the decision of which processor P creates. Initially, the delegate creates instances of P. Then, when the variant v_{n+1} is added and P is extended to P', a new delegate overrides the old one to create instances of P' instead. We can encode this idea to create the *Extensible Visitor* protocol as follows:

1. The creation of new processors is performed via a separate method: a *virtual constructor* (or Factory Method [13]), called *makeContainsPt* in our example.

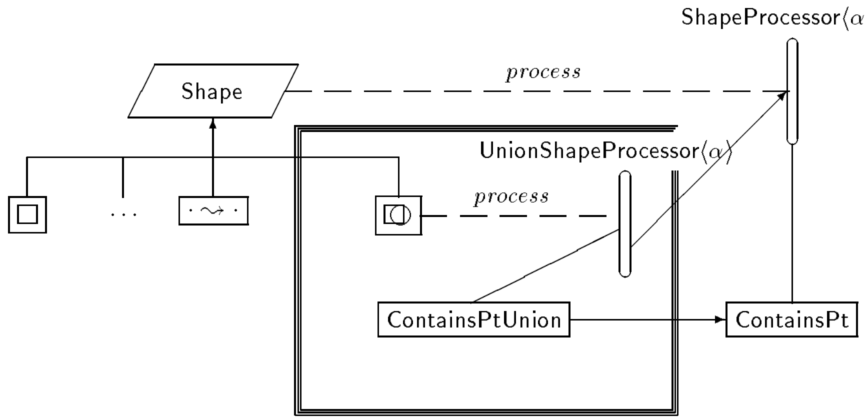

Figure 13: Datatype and Processor Extension

2. The virtual constructor is an η-expansion of the original constructor, *e.g.*, in ContainsPt, the virtual constructor is

   ContainsPt *makeContainsPt* (Point p) {
   return new ContainsPt (p) ; }

3. Expressions that construct processors are replaced with invocations of the virtual constructor.

4. The virtual constructor is overridden in all extensions of processors. Thus, in ContainsPtUnion, we now have

   ContainsPt *makeContainsPt* (Point p) {
   return new ContainsPtUnion (p) ; }

The final version of the code is shown in Fig. 12.

The form of the system after the extension is shown in Fig. 13. The rectangles represent concrete classes, the parallelogram an abstract class, and the thin ovals interfaces. Solid lines with arrowheads show inheritance, while those without arrowheads indicate that a class implements an interface. Dashed lines connect classes and interfaces. The label on a dashed line names a method in the class that accepts an argument whose type is the interface. The boxed portion is the extended datatype and its corresponding processor. For a processor and datatype extension all code outside of the thick box can be re-used without any change.

3.5 Updating Dependencies Between Tools and Datatypes

The problem of updating the dependencies of processors has a general counterpart. Suppose the processors P_1, P_2, and P_3 all process the same datatype D and depend on each other as follows: P_1 creates instances of P_1 and P_2, P_2 uses P_3, and P_3 uses itself. Figure 14 (a) illustrates this situation: each processor at the tail of an arrow creates an instance of the processor at the head. When D is extended to D' with new variants, the tools are extended to P_1', P_2', and P_3', respectively. If P_1, P_2, and P_3 directly create instances of each other, however, the extensions cannot process all of D'.

This problem can be resolved with the following extension of Extensible Visitor. Each processor, P, is equipped with a virtual constructor for every processor that it uses (including itself). This is shown in Fig. 14 (b), where the dashed lines indicate the use of a virtual constructor to create

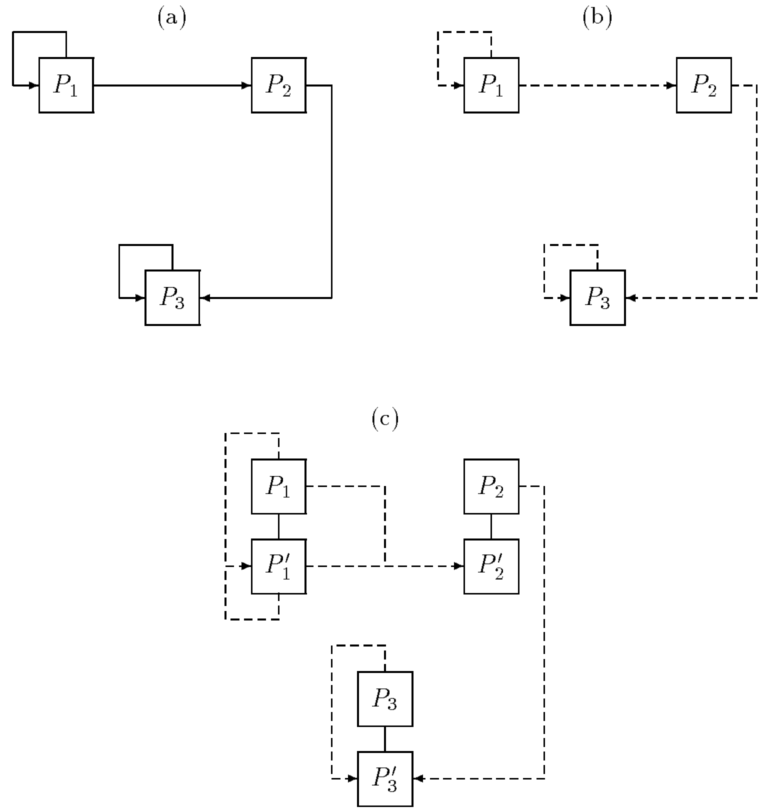

Figure 14: Updating Dependencies Between Tools

instances of processors. When P is extended to reflect a datatype extension, every virtual constructor is correspondingly overridden. Thus each processor gets the most current version of the tools it uses (see Fig. 14 (c)) while existing code remains unchanged. In the example, all the existing dependencies are redirected, and two new ones are added: P_1' on itself and on P_2'.

A related problem arises when a program contains datatypes which are mutually recursive. Consider a multimedia editor that supports both text and images. Suppose we wish to incorporate our graphical package into the editor. The editor provides a new kind of shape, Hybrid, which contains a Drawable element. Each Drawable entity is either a Char or an Image, and each Image contains a Shape. Thus, Shape and Drawable are mutually recursive. Figure 15 shows these new definitions.

Figure 16 presents two processors, RenderShape and RenderDrawable, which take a display device as an argument and render Shapes and Drawables on the device, respectively. Each processor uses a virtual constructor to create new instances of itself and of the processor for the other datatype. An extension of a datatype requires an upgrade of both processors. Their virtual constructors for the processor corresponding to the datatype before extension must now create instances of the processor that accepts the extended datatype. In short, we can treat these two processors as if they were unrelated (rather than implementing the same functionality over two related datatypes), and redirect their dependencies as discussed above.

4 Types

Typed object-oriented languages can provide (at least) two kinds of polymorphism: *object polymorphism* and *parametric polymorphism*. Object polymorphism means that a variable declared to be of a particular class (type), say C, can hold instances of C or subclasses of C. In contrast, parametric polymorphism allows types to contain type variables that are (implicitly) universally quantified; for example, $list(\alpha)$ is the type of a homogenous list containing any type of element. Most typed object-oriented languages provide object polymorphism; a few offer parametric polymorphism.[6]

Pizza's parametric polymorphism greatly facilitates the implementation of Extensible Visitors.[7] To illustrate this point in more detail, we contrast the Pizza implementation with one in Java. In Java, if *process* is expected to return values, its return type must be declared as Object. Choosing any other type C_p would force all clients to return subtypes of C_p, which is inappropriate for some clients and prevents re-use of existing libraries and classes.[8] All clients that invoke processors—including recursive invocations—must then use narrowing casts to restore the returned value to its original type. If we translate ContainsPt to return Boolean instead of **boolean**, the Java version of the *forUnion* method in ContainsPtUnion is:

```
public Object forUnion ( u) {
  return new Boolean
    ((((Boolean) (u.lhs.process (this))).booleanValue ()) ∨
      (((Boolean) (u.rhs.process (this))).booleanValue ())) ; }
```

[6] C++'s [31] template mechanism provides a limited amount of parametric polymorphism.

[7] Thorup [33] has proposed a different style of type parameterization for Java: virtual types. To implement Extensible Visitor using virtual types, which are overrideable types in classes analogous to virtual methods, and obtain the benefits of type-checking, we need to declare *process* as follows (where α is the virtual type declared in the processor):

$$p.\alpha \ process \ (\mathsf{ShapeProcessor}\ p)$$

Unfortunately, this is currently not possible with virtual types [personal communication, August 1997]. Hence, virtual types are not yet a viable alternative for our Extensible Visitor.

[8] The choice of Object still cannot accommodate processors (such as ContainsPt) that return primitive types, which are not subtypes of any other type, including Object [14]. Such processors are forced to use the "wrapped" versions of primitive types, incurring both space and time penalties.

```
class Hybrid extends Shape {
   Drawable d;
   ⋮ }

abstract class Drawable { ··· }
class Char extends Drawable { ··· }
class Image extends Drawable {
   Shape s;
   ⋮ }
```

Figure 15: Mutually Recursive Datatypes

```
class RenderShape implements ShapeProcessor⟨void⟩ {
   Device d;
   RenderShape (Device d) { this.d = d ; }
   RenderShape makeRenderShape (Device d) {
      return new RenderShape (d) ; }
   RenderDrawable makeRenderDrawable (Device d) {
      return new RenderDrawable (d) ; }
   ⋮ }

class RenderDrawable implements DrawableProcessor⟨void⟩ {
   Device d;
   RenderDrawable (Device d) { this.d = d ; }
   RenderDrawable makeRenderDrawable (Device d) {
      return new RenderDrawable (d) ; }
   RenderShape makeRenderShape (Device d) {
      return new RenderShape (d) ; }
   ⋮ }
```

Figure 16: Tools over Mutually Recursive Datatypes

For the Pizza version of the same code (see Fig. 11) the compiler statically verifies that the return type of a processor is acceptable in each invoking context. Thus, in a proper implementation, the programmer gets the full benefit of type-checking, and the program incurs no runtime expense. In contrast, the Java version passes the type-checker, but the programmer is forced to specify runtime checks. These checks compromise both the program's robustness and its efficiency. A Java compiler could eliminate some of these checks, but this would rely on sophisticated flow analyses, which few compilers (if any) perform.[9]

Even Pizza requires the programmer to repeat several pieces of type information. For example, when ContainsPtUnion is defined as an extension to ContainsPt, the type parameter of UnionShapeProcessor must still be instantiated (see Fig. 12). Also, the methods inside a processor need type declarations, even though the return type is the same as the parameter of the interface. A powerful type inference mechanism, such as those of Eifrig, Smith, and Trifonov [7] and Palsberg [22], can alleviate many of these problems, especially in the context of dynamically-typed object-oriented languages.

5 A Language for Extensible Systems

Although the Extensible Visitor pattern solves our problem, it requires the management of numerous mundane details, such as writing class declarations to define the datatype and its variants, defining and overriding the virtual constructors, and keeping the type information consistent. Since these tasks are cumbersome and error-prone and can be managed automatically, we have also designed and implemented a language extension for specifying instances of the Visitor and Extensible Visitor patterns.

Our system, called Zodiac, provides constructs for declaring and extending datatypes and processors. Datatypes and processors are translated into collections of classes. Processors are defined with respect to a datatype. The action for each variant V of the datatype is implemented by a method m_V in the processor. The method m_V accepts one argument, which is an instance of the class used to implement the variant V.

Figure 17 illustrates how to use a Pizza-oriented version of Zodiac to specify the datatype and toolkit for our running example. At the top we define the collection of shapes, followed by the ContainsPt processor. Below that we specify UnionShape, which is Shape extended with the union of two shapes, and its corresponding processor as an extension of ContainsPt. The example uses all of Zodiac's constructs:

datatype defines a new extensible datatype or **extends** an existing one.[10] Each variant of the datatype, together with its fields, is listed following the keyword **variant**. Zodiac creates an abstract class for a new datatype, and translates each variant into a concrete subclass with a *process* method.

processor defines a processor for the datatype that is specified in the **processes** clause. The (optional) **uses** clause is followed by a list of tools that are used by the processor.[11] The processor's return type is declared after **returns**. The (optional) **fields** clause specifies the parameters of a processor, from which Zodiac determines the instance variable declarations and the constructor. The individual methods for the variants are declared with **variant**.

Zodiac creates a virtual constructor, such as *makeContainsPt* in the example, for each tool listed as a dependency. Processor extensions inherit the **returns** and **fields** declarations and

[9] These comments apply equally well to the Visitor protocol.

[10] This datatype construct is superficially related to Pizza's algebraic data types. Pizza's data types are meant principally for creating data structures; they do not provide default visitor methods.

[11] This clause is optional since a tool may not have any dependencies to declare. This information cannot be inferred since deciding which tool dependencies should be updated is a design decision that must be made by the programmer.

```
datatype Shape {
   variant □ (double s);
   variant ○ (double r);
   variant · ⇝ · (Point d, Shape s);
}

processor ContainsPt processes Shape
                     uses ContainsPt
                     returns boolean {
   fields (Point p);
   variant for□(s) { ··· }
   variant for○(c) { ··· }
   variant for· ⇝ ·(t) {
      return t.s.process (makeContainsPt (···)) ; }
}

datatype UnionShape extends Shape {
   variant ⧉ (Shape lhs, Shape rhs);
}

processor ContainsPtUnion extends ContainsPt
                          processes UnionShape {
   variant for⧉(u) {
      return u.lhs.process (this) ∨
             u.rhs.process (this) ; }
}
```

Figure 17: Sample Extended Pizza Specification

the **uses** dependency of their parent. A derived processor needs to declare only the new fields and dependencies. The constructor of a processor extension accepts values for all its fields and those of its superclass, and conveys values for the inherited fields to its superclass's constructor.

Zodiac expands the Extensible Visitor specification into a collection of classes and interfaces that is α-equivalent to the code in Sect. 3.

6 Implementation and Performance

Zodiac is currently implemented as a language extension to MzScheme [12], a version of Scheme [3] extended with a Java-like object system.

A preliminary version of Zodiac has been used to implement DrScheme, a Scheme programming environment [11]. DrScheme is a pedagogically-motivated system that helps beginners by presenting Scheme as a succession of increasingly complex languages. It also supports several tools such as a syntax checker, a program analyzer, *etc.*

The largest language handled by DrScheme is the complete MzScheme language, which is many times the size of standard Scheme. Still, the language processing portions of DrScheme were developed and are maintained (part-time) by a single programmer. The preliminary implementation of Zodiac played a significant rôle in this rapid development. It simplified the specification of the language tower, which, in turn, avoided many clerical errors and facilitated the maintenance of the software.

Our current implementation has been in use for about two years. The resulting environment is used daily in courses at Rice University and other institutions. The environment is also used to develop actual applications, and the overhead of Extensible Visitor is low enough to be practical for such use.

Zodiac is also being applied in other domains. We have used it to build Chisel, a general-purpose, extensible document construction system. This system handles "real-world" documents, and easily meets demanding performance criteria. For example, Chisel generates our entire departmental brochure (corresponding to 20-30 printed pages, or about 150 kilobytes of generated HTML) in 20 seconds on a modern workstation.

The marginal cost of using our method over the Visitor pattern is minimal. The sole difference is in the creation of processors. When the virtual constructor is not overridden, the only cost is that of a local method call, which is effectively inlined in Visitor. In many cases this overhead is avoided entirely because the current instance is re-used for recursive calls. The overall cost of this indirection depends on how often an application constructs data, and on the implementation model used for objects and methods. In our experience, this cost has been negligible.

7 Background and Related Work

Several researchers, including Cook [4], Kühne [18], Palsberg and Jay [23], and Rémy [26], have observed the trade-offs between the functional and object-oriented design approaches, and have noted the relative strengths and weaknesses of each method at datatype and toolkit extension. Of them, only Kühne [18] and Palsberg and Jay [23] suggest a solution.

Kühne's solution [18] is to replace the dispatching in the Visitor protocol with generic functions that perform double-dispatch. While Kühne's approach can accommodate legacy classes, *i.e.*, classes that do not have an explicit method for the visitor, it has the disadvantage of potentially violating the hierarchical design of the program, does not address the organization of the generic function itself, and depends on language features that support double-dispatch.

Palsberg and Jay [23] propose to use reflection to implement a Visitor-like protocol. In their protocol, all visitors are subclasses of the Walkabout class, which provides a default visitor. The default visitor examines the argument; if the argument is not a base class, the Walkabout obtains the argument's fields using Java's reflection facility [32], and then recursively visits each field.

While Palsberg and Jay's approach also scales to legacy classes, it is unclear how well their system works when the variants have instance variables unrelated to the fields of the variant, or when they have multiple fields with the same type. Their proposal also relies on the existence of reflective operators, which are not found in many languages. Finally, their system is over two orders of magnitude slower than a plain Visitor, making it unsuitable for practical use. In contrast, Extensible Visitor works with generic object-oriented languages, and incurs a negligible overhead beyond that of Visitor.

Lieberherr and his colleagues have built a system for adaptive programming [24], which addresses the structural and behavioral adaptation of systems. Using their system, Demeter, programmers write separate specifications of traversals and actions, and Demeter combines these to generate a complete program. In particular, Demeter consumes four inputs: a description of the class graph, a traversal specification for the graph, the operations to perform at each node, and some glue code for linking traversals and operations. Consequently, Demeter is only applicable when all these specifications are available for the production team to reconstruct the program. A company that wishes to distribute its product only in the form of object code to protect its proprietary algorithms would probably be unwilling to distribute its Demeter specification. In contrast, our method both assumes an open-ended program and allows the distribution and extension of object code.

The literature on design patterns contains many other attempts to define and implement patterns similar to Interpreter and Visitor. The primary presentation of the Visitor pattern [13] states that datatype extension is difficult, but does not solve the problems that arise. Baumgartner, Läufer and Russo [1] propose an implementation of Visitor based on multi-method dispatch and claim that it makes datatype and toolkit extension easy, but they do not recognize the problems that arise when extending tools or coordinating multiple tools. Seiter, Palsberg, and Lieberherr [29] describe how dynamic relationships between classes can be captured more expressively using *context relations*, which extend and override the behavior of classes and decouple behavioral evolution and inheritance hierarchies. While context relations offer a more concise way of expressing Visitor-like operations, the authors do not mention or solve the recursive instantiation problem (described in Sect. 3.3).

We can alternatively view the variants of a datatype as specifying the terms of a language, and interpreters as tools. The functional language community has been interested in the problem of creating interpreters from fragments that interpret portions of the language [2, 8, 19, 30]. These approaches are orthogonal to ours in that they can handle semantic extensions to the interpreters, but none of them consider the problem of an extensible toolkit. Most of them [2, 8, 30] do not address the problem of extending the datatype either.

Duggan and Sourelis [6], Findler [10], and Liang, Hudak, and Jones [19] describe methods for creating restricted notions of extensible datatypes. None of these approaches, however, produce datatypes that are extensible in the sense of our protocol. The programmer may specify variants of the datatype separately, but the final datatype must be assembled and "closed" before it can be used. As a result, it is not possible to extend the variants of an existing datatype. Any further additions require access to the source code.

Cartwright and Felleisen's work on extensible interpreters [2], if translated into an object-oriented framework, would probably resemble the Extensible Visitor protocol in an untyped setting.

8 Conclusions and Future Work

We have presented a programming protocol, Extensible Visitor, that can be used to construct systems with extensible recursive data domains and toolkits. It is a novel combination of the functional and object-oriented programming styles that draws on the strengths of each. The object-oriented style is essential to achieve extensibility along the data dimension, yet tools are organized in a functional fashion, enabling extensibility in the functional dimension. Systems based on the Extensible Visitor can be extended without modification to existing code or recompilation (which is an increasingly important concern).

We have also described Zodiac, a language extension for writing extensible programs. Zodiac manages the mundane and potentially error-prone administrative tasks that arise when implementing the Extensible Visitor. A variant of Zodiac has been in use for about two years in our programming environment DrScheme [11]. Through it, DrScheme is able to offer a hierarchy of language levels that facilitate a pedagogically sound introduction to programming. It supports multiple program-processing tools that operate over this range of language levels. Zodiac has also been used to build other systems, such as a document generator with multiple rendering facilities.

Our work suggests future investigations into the efficiency of the new language facilities. The current implementation of Extensible Visitor incurs an execution penalty due to dispatching. Indeed many design patterns suffer similar overheads, but their popularity suggests that users are more interested in design and extensibility considerations than in fine-grained efficiency. For example, Portner [25] reports that his use of the Interpreter pattern to implement a command language is up to 30% slower than a hand-crafted C implementation; still, he states that the low development cost far outweighs the execution penalty. Nevertheless, we believe that a compiler can exploit a Zodiac specification and assemble more efficient code than the naïve translation outlined above.

Acknowledgments

We thank Corky Cartwright, Mike Fagan, Bob Harper, Thomas Kühne, Karl Lieberherr, Jens Palsberg, and Scott Smith for helpful discussions and for comments on preliminary versions of this paper.

References

[1] Baumgartner, G., K. Läufer and V. F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, Feburary 1996.

[2] Cartwright, R. S. and M. Felleisen. Extensible denotational language specifications. In Hagiya, M. and J. C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Science*, pages 244–272. Springer-Verlag, April 1994. LNCS 789.

[3] Clinger, W. and J. Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[4] Cook, W. R. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, pages 151–178, June 1990.

[5] Coplien, J. O. and D. C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

[6] Duggan, D. and C. Sourelis. Mixin modules. In *ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, May 1996.

[7] Eifrig, J., S. Smith and V. Trifonov. Type inference for recursively constrained types and its application to OOP. *Mathematical Foundations of Program Semantics*, 1995.

[8] Espinosa, D. Building interpreters by transforming stratified monads. Unpublished manuscript, June 1994.

[9] Felleisen, M. and D. P. Friedman. *A Little Java, A Few Patterns*. MIT Press, 1998.

[10] Findler, R. B. Modular abstract interpreters. Unpublished manuscript, Carnegie Mellon University, June 1995.

[11] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs*, 1997.

[12] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[13] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.

[14] Gosling, J., B. Joy and G. L. Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.

[15] Hudak, P. and M. P. Jones. Haskell *vs.* Ada *vs.* C++ *vs.* Awk *vs.* ... An experiment in software prototyping productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT, USA, October 1994.

[16] Hudak, P., S. Peyton Jones and P. Wadler. Report on the programming language Haskell: a non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992. Version 1.2.

[17] Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[18] Kühne, T. The translator pattern—external functionality with homomorphic mappings. In *Proceedings of TOOLS 23, USA*, pages 48–62, July 1997.

[19] Liang, S., P. Hudak and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, 1992.

[20] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[21] Odersky, M. and P. Wadler. Pizza into Java: Translating theory into practice. In *Symposium on Principles of Programming Languages*, pages 146–159, Janurary 1997.

[22] Palsberg, J. Efficient inference of object types. *Information & Computation*, 123(2):198–209, 1995.

[23] Palsberg, J. and C. B. Jay. The essence of the Visitor pattern. Technical Report 05, University of Technology, Sydney, 1997.

[24] Palsberg, J., C. Xiao and K. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, 1995.

[25] Portner, N. Flexible command interpreter: A pattern for an extensible and language-independent interpreter system, 1995. Appears in [5].

[26] Rémy, D. Introduction aux objets. Unpublished manuscript, lecture notes for *course de magistère*, Ecole Normale Supérieure, 1996.

[27] Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Schuman, S. A., editor, *New Directions in Algorithmic Languages*, pages 157–168. IFIP Working Group 2.1 on Algol, 1975.

[28] Riehle, D. Composite design patterns. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 218–228, 1997.

[29] Seiter, L. M., J. Palsberg and K. J. Lieberherr. Evolution of object behavior using context relations. *IEEE Transactions on Software Engineering*, 1998.

[30] Steele, G. L., Jr. Building interpreters by composing monads. In *Symposium on Principles of Programming Languages*, pages 472–492, Janurary 1994.

[31] Stroustrup, B. *The C++ Programming Language.* Addison-Wesley, 1991.

[32] Sun Microsystems. Java core reflection. API and Specification, 1997.

[33] Thorup, K. K. Genericity in Java with virtual types. In *European Conference on Object-Oriented Programming*, pages 444–471, 1997.