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

Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman*
Department of Computer Science
Rice University

Contact: shriram@cs.rice.edu

December 1, 1997

Summary

Current programming practice frequently confronts programmers with the following design dilemma. A problem requires a recursively specified type of data and a collection of tools on that type. The problem is expected to evolve over time in different ways, or different variants of the problem arise. The programmer should therefore construct the program in a way that makes it easy to

1. extend the types and adjust the existing tools accordingly, and
2. extend the toolkit.

Ideally, this evolution should not rely on modifications to existing code. First, modification is cumbersome and error-prone. Second, the source may not be available. Third, the program may need to evolve in several different directions, making modification expensive. Finally, modification dissolves abstraction boundaries and violates the spirit of re-use.

Unfortunately, the prevailing functional and object-oriented design strategies do not support both forms of extensibility. While the former accommodates only the addition of new tools, the latter supports either adding new tools or extending the underlying data set but not both. Neither style accommodates both forms of extension.

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 strategies. We also show how this protocol suggests a new set of linguistic facilities for languages that support class systems.

*Permanent address: Computer Science Department, Indiana University, Bloomington, IN 47405.

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 changes, 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.

In addition, 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 sold in the form of object code, possibly over the Web. Third, it may be necessary to evolve the base program in several different directions, in which case code modifications are prohibitively expensive because the required duplication would result in duplicated maintenance costs. Finally, source modification dissolves the abstraction boundary of the original system, and violates the spirit of re-use.

This dilemma manifests itself in many different application areas. A particularly important example arises when processing programming languages. Language grammars are typically specified via BNFs, which denote recursively defined 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 implemented 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 [11], this approach is known as the Interpreter pattern. It suffers from the dual problem of the functional one: variants are easy to add, while tool additions require code modification.

  If the collection of tools is large, the designer may also use the Visitor pattern, a variant of the Interpreter that collects the code for a tool in a single class. This choice 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 [25] that 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

[1] Rémy [23] describes the problem in a concise manner; the problem was first anticipated by Reynolds [24].

a meta-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; the abstraction interpreter 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, and Section 6 discusses our implementation and experience. The last two sections describe related work and summarize the ideas in this paper.

2 Existing Techniques

To illustrate the problem with a concrete example, we present a simplistic "geometry manager" program, derived from a US Department of Defense programming contest [13]. We discuss both the functional and the object-oriented design methods in this context and expose their failings. We use the term *tool* to refer to a service offered by the complete system, and *processor* for a concrete class or function that implements a tool.

Initially, our system specifies a datatype (Shape) with three variants—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 (PointIn). The set of shapes is then extended to include a composite shape that is the union of two others ($\square\!\bigcirc$). The set of tools grows to include one that, given a number and a shape, creates a copy of that shape shrunken in area 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 tagged *variants*. In SML [18] or Haskell [14], for example, a programmer could use the **datatype** or **data** construct, respectively, to represent the set of shapes, as shown in Figure 1.[2] Each variant declares the types of its fields, which can 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 circle by its radius, and a translated shape by a displacement 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) whose side has length 3.

Processors map variants of the Shape datatype to results. For example, Figure 2 shows the outline of the processor PointIn—whose mathematics is elided because it is irrelevant—

[2] In C [15], one would use (recursive) pointers, structs and unions to represent this set of constructs.

```
datatype Shape = □ of num
               | ◯ of num
               | · ⇝ · of Point × Shape
```

Figure 1: The Functional Approach: Types

```
PointIn : Point × Shape ⟶ boolean

PointIn p (□ s) = ···
      | p (◯ r) = ···
      | p (· ⇝ · d s) = ··· PointIn p' s ···
```

Figure 2: The Functional Approach: Tools

```
Shrink : num × Shape ⟶ Shape

Shrink pct (□ s) = (□ ···)
      | pct (◯ r) = (◯ ···)
      | pct (· ⇝ · d s) = (· ⇝ · d (Shrink pct s))
```

Figure 3: The Functional Approach: Adding Tools

which determines whether a point is inside a shape. The function definition uses pattern-matching: if a pattern matches, the identifiers to the left of the = are bound on the right to the corresponding inputs. For example, the pattern (□ s) in the first line of each 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 processors are (tail) recursive, too. The recursive calls in a processor match the recursive structure of the type. A similar template can be used to define other processors (Figure 3); there is no need to modify any existing code for this addition.

In the functional style, the core 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 that handles the variants.

Unfortunately, it is impossible to add a variant to Shape without modifying existing code in two ways. 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 [5, 16, 18]. Second, even if extensible datatype definitions are available, the code for each processor, such as PointIn, must be edited to accommodate these extensions to the datatype.

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

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

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

and provide implementations of the actual operations. This is known as the Interpreter pattern [11].[3] For instance, the SML program from Figures 1 and 2 corresponds to the Java [12] program shown in Figure 4. The recursive references among instance variables and classes lead to corresponding recursive calls among methods, analogous to the recursion in the functional program.

Extending the set of shapes is straightforward. It suffices to add a new concrete class that extends Shape and whose methods specify the behavior of the existing processors for that extension. For example, Figure 5 shows how the union of two shapes, denoted ◻◯, can be added as a shape to our system. Existing processors do not need to be modified.

Unfortunately, the Interpreter pattern is unsuitable when a new tool must be added. If existing code is not supposed to be changed, the only option is to create, for each concrete class, an extension that defines a method for the new tool. This requires extending every existing client to create instances of the new, extended classes instead of the old ones. This change also affects existing clients whose output must be processed by the old and new processors.

An existing processor may be one such client. For example, in Figure 6, the *shrink* methods create concrete instances of Shape that have access to only the *pointIn* and *shrink* methods (tools). If a processor added later uses *shrink*, the returned object will not support all operations—unless the *shrink* method is updated.

In summary, object-oriented programming—as represented by the Interpreter pattern—provides the equivalent of an extensible, user-defined datatype, which addresses 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

[3] The Composite pattern [11] is sometimes used instead.

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

Figure 5: The Object-Oriented Approach: Adding Types

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

Figure 6: The Object-Oriented Approach: Adding Tools

processor must reside in Shape (unless the language has multiple-inheritance), even though the abstraction may not apply to most tools and hence does not belong in Shape.

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 extensibility as well as re-use along both dimensions are essential.

In this section, we develop a programming protocol based on object-oriented concepts that satisfies our stated desiderata. We present the protocol in three main stages. First we explain how to represent extensible datatypes and tools via the Visitor pattern [11] and how the Visitor pattern suffers from the same problem as the functional design strategy. Still, the Visitor protocol can be modified so that a programmer can extend the datatype and tools in a systematic manner. Finally, we demonstrate how the protocol can accommodate extension across multiple datatypes and tools.

The ideas are illustrated with fragments of code written in Pizza [19], a parametrically polymorphic extension of Java. The choice of Pizza is explained in Section 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 similar to that of the Interpreter pattern, except that each class (variant) contains only one interpretive method per variant: *process*. This method consumes a *processor*, dispatches to a method from the processor that corresponds to the variant, and returns the result of the invoked method. Figure 7 illustrates how the datatype from Section 2.2 is represented according to this protocol.

Since different processors return different types of results, the type of *process* is best

```
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

represented as a parametrically polymorphic type. More specifically, in our example, *process* has the type ShapeProcessor$\langle\alpha\rangle \longrightarrow \alpha$. That is, *process*'s argument has the parametric type ShapeProcessor$\langle\alpha\rangle$, which is implemented as an interface in Pizza. The return type of *process* is α, the type parameter of the processor. In Pizza, this is denoted by $\langle\alpha\rangle\ \alpha$ in place of a single, fixed type for the range of the method.

It follows from the description above that a processor must specify one method per variant in the datatype. For our running example, the parametric interface and the outline of the point containment-checking tool (PointIn) are shown in Figure 8.

Processors that depend on parameters other than the datatype instance being processed accept these as arguments to their constructor and store them in instance variables. Thus, to check whether a point p is in a shape s, we create an instance of PointIn, which is of type ShapeProcessor$\langle$**boolean**$\rangle$, with the point p as an argument. This instance of PointIn is passed to the shape's *process* method:

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

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. In particular, the PointIn tool deals with translated shapes by translating the point and checking it against the underlying shape. This is shown in Figure 8. The underlined expression implements the creation of a new processor.

The Visitor pattern ensures that the code for each tool is localized and easily comprehensible, as in the functional approach.

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

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

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

3.2 Adding Tools

The Visitor pattern facilitates the extension of the tool collection. For instance, a processor that shrinks shapes would implement the ShapeProcessor⟨Shape⟩ interface. This is outlined in Figure 9. In this example, a translated shape is shrunk by shrinking the underlying shape; the shrink factor does not change. Hence, the recursive call uses the same processor (**this**, underlined in the figure).

Other tools can be added in a similar fashion.

3.3 Extending the Datatype: A False Start

Since concrete sub-classes represent the variants of a datatype, extending a datatype description means adding new concrete sub-classes. The only requirement is that 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[4] specifies one method per variant in the old datatype and one for each new variant. Of course, the *process* method in the new datatype variants

[4] We think of an interface as the collection of method signatures specified in an **interface** clause and all those specified in the super-interfaces.

```
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 PointInU extends PointIn implements UnionShapeProcessor⟨boolean⟩ {
  PointInU (Point p) { super (p) ; }
  public boolean forUnion (◫ u) {
    return u.lhs.process (this) ∨ u.rhs.process (this) ; } }
```

Figure 11: Processor Extension

should only accept processors that satisfy the new interface. This requirement is expressed differently in different languages; in Pizza, for example, we use a run-time check.

To illustrate this idea, we add the union shape (◫) to the collection of shapes. The new concrete class and interface are shown in Figure 10. A cast (underlined in the figure) requires the processor for ◫'s to implement the extended interface, UnionShapeProcessor. The extended processors can 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 Figure 11.

Unfortunately, this straightforward extension of PointIn is incorrect. Consider the *forTranslated* method in PointIn. It creates a new instance of PointIn to process the underlying shape with respect to the translated point (underlined in Figure 8). Since PointIn does not include a *forUnion* method, it cannot process the language extension. Thus, a shape with a union shape (◫) inside one of the original shapes, such as

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

(the union of a square and a circle translated to p), causes evaluation to halt with an error when the processor created in *forTranslated* is called upon to process a union (◫).

3.4 Extending the Datatype: The Solution

The error points out that processors in the Visitor pattern are not designed to accommodate extensions of the datatype. Suppose a processor P can handle the variants $v_1, \ldots, v_n$ and invokes itself to compute an answer. As long as the recursive call uses **this**, substituting a subtype of P for P works fine. If, however, P creates a new instance specifically of P, it can only be used for the variants $v_1, \ldots, v_n$. When a new variant, v_{n+1}, is added, the recursive call to P can no longer process all possible inputs.

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

class PointInU extends PointIn implements UnionShapeProcessor⟨boolean⟩ {
  PointInU (Point p) { super (p) ; }
  PointIn makePointIn (Point p) {
    return new PointInU (p) ; }
  public boolean forUnion (□◯ u) {
    return u.lhs.process (this) ∨ u.rhs.process (this) ; } }
```

Figure 12: Extensible Visitor Processor Extension

We can avoid this premature specification of a recursive call by abstracting over the processor that P creates. Then, when the variant v_{n+1} is added and P is extended to P', the abstraction can be overridden to create an instance of P' instead. We can encode this idea in the protocol as follows:

1. The creation of new processors is performed via a separate method: a *virtual constructor* (or Factory Method [11]), called *makePointIn* in our example.

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

```
PointIn makePointIn (Point p) {
  return new PointIn (p) ; }
```

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

4. The virtual constructor is overridden in all processor extensions. Thus, in PointInU, we now have

```
PointIn makePointIn (Point p) {
  return new PointInU (p) ; }
```

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

The form of the system after the extension is shown in Figure 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

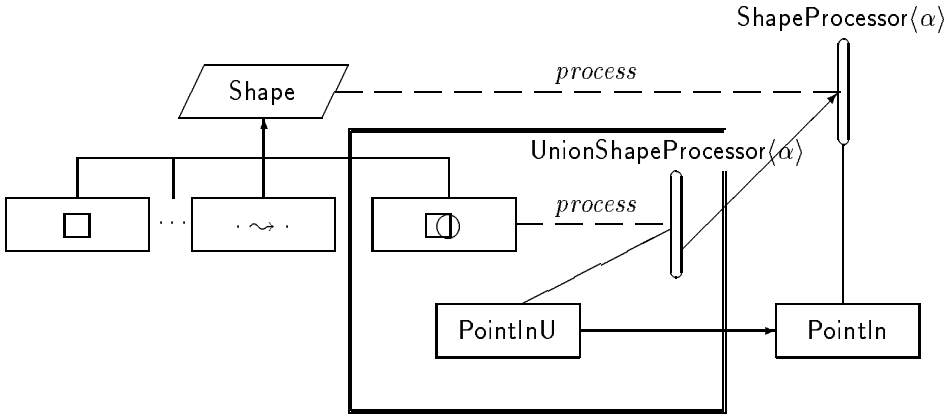

Figure 13: Datatype and Processor Extension

interface. The boxed portion is the extended datatype and its corresponding processor. For a processor and datatype extension all code outside of the box can be used without any change.

3.5 Updating Dependencies Between Tools

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. However, if P_1, P_2 and P_3 directly create instances of each other, the extensions cannot process all of D'.

This problem can be resolved with the following extension of the protocol. Each processor, P, is equipped with a virtual constructor for every processor that it uses (including itself). This is shown in Figure 14 (b), where the dashed lines indicate the use of a virtual constructor to create instances of processors. When P is extended to reflect a language extension, every virtual constructor is correspondingly overridden. Thus each processor gets the most current version of the tools it uses (see Figure 14 (c)) while existing code remains unchanged. In the example, all the existing dependencies are updated, and two new ones are added: P_1' on itself and on P_2'.

This protocol extends naturally to multiple datatypes.

3.6 Performance Considerations

Our protocol incurs negligible execution overhead beyond that of the Visitor pattern. The sole difference is in the creation of processors at recursive calls. Whereas the Visitor pattern creates the processor directly, ours requires an indirection through a method invocation. In many cases this overhead is avoided entirely because the current processor is used for

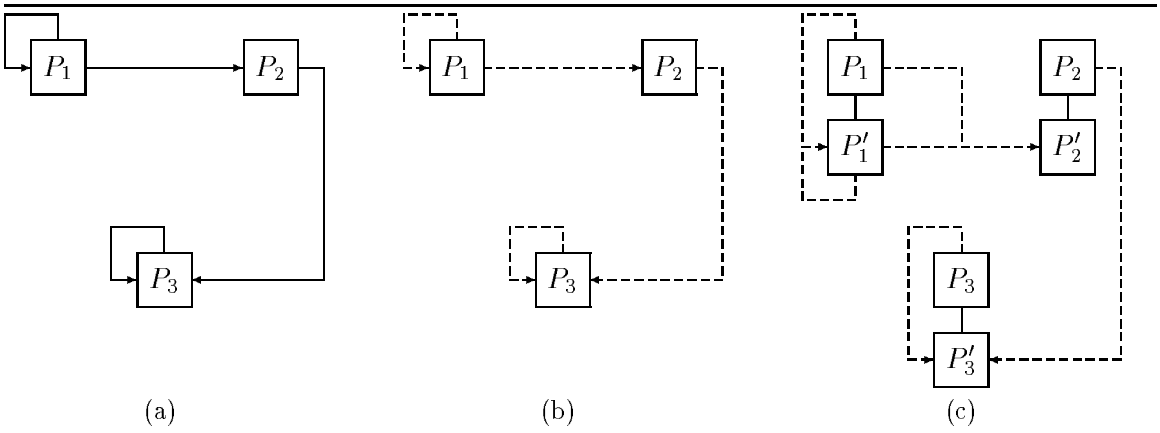

Figure 14: Updating Dependencies Between Tools

recursive calls. In our experience, the resulting cost in time is within acceptable bounds.

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 sub-classes of C. In contrast, parametric polymorphism allows types to contain type variables that are universally quantified. Most typed object-oriented languages provide object polymorphism; a few offer parametric polymorphism.

Pizza's parametric polymorphism greatly facilitates the implementation of our protocol.[5] To illustrate this point in more detail, we contrast the Pizza implementation with one in Java. In Java, *process*'s return type must be declared as Object. Choosing any other type C_p would force all clients to return sub-types of C_p, which may not be appropriate for some clients, and prevents re-use of existing libraries and classes.[6] All clients that use the result from processors—including recursion in processors—must then use narrowing casts to restore the value to its original type. If we translate PointIn to return Boolean instead of **boolean**, the Java version of the *forUnion* method in PointInU is:

[5] Thorup [27] has proposed a different style of type parameterization for Java: virtual types. To implement our protocol 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 in general with virtual types [personal communication, August 1997]. Hence, virtual types are not yet a viable alternative for our protocol.

[6] The choice of Object prevents processors (such as PointIn) from returning primitive types, which are not subtypes of Object [12]. Such processors are forced to use the "wrapped" versions of primitive types, incurring both space and time penalties.

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

For the Pizza version of the same code, shown in Figure 11, the compiler statically verifies that the type returned by a processor is acceptable to its invoking context. Thus, in a proper implementation, the programmer gets the full benefit of type-checking, and the program incurs no run-time expense. In contrast, although the Java version type-checks, the programmer is forced to specify run-time checks. This compromises 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.

Even Pizza requires the programmer to repeat several pieces of type information. For example, when PointInU is defined as an extension to PointIn, UnionShapeProcessor must still instantiate the type parameter. 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 [6] and Palsberg [20], can alleviate many of these problems.

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 meta-language 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, which are defined with respect to a datatype, provide one method per variant. Each method accepts one argument, which is an instance of (the class representing) the variant that is processed by that method.

Figure 15 illustrates how to use a Pizza-based version of Zodiac to specify the datatype and toolkit for our running example. In the left column we define the collection of shapes and the PointIn processor. In the right column we specify UnionShape, which is Shape extended with the union of two shapes, and its corresponding processor as an extension of PointIn. The example uses all of Zodiac's language extensions:

datatype defines a new extensible datatype or **extends** an existing one. 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 in a new or extended datatype into a concrete sub-class with a *process* method. The body of the methods is as described in Section 3.

processor defines a processor for the datatype specified in the **processes** clause. The (optional) **uses** clause is followed by a list of the other tools that the one being defined

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

processor PointIn processes Shape
                  uses PointIn
                  returns boolean {
  fields (Point p);
  variant for□(s) {
    ··· }
  variant for◯(c) {
    ··· }
  variant for· ⇝ ·(t) {
    return
      t.s.process (
        makePointIn (···)) ; }
}
```

```
datatype UnionShape extends Shape {
  variant □◯ (Shape lhs, Shape rhs);
}

processor PointInU extends PointIn
                   processes UnionShape {
  variant for□◯(u) {
    return u.lhs.process (this) ∨
           u.rhs.process (this) ; }
}
```

Figure 15: Sample Extended Pizza Specification

depends on. The processor's return type is declared after **returns**. The **fields** clause specifies the additional parameters of a processor; it is replaced by instance variable declarations and an appropriate constructor. The individual methods for the variants are declared with **variant**. A virtual constructor, such as *makePointIn* in the example, is automatically defined for each tool listed as a dependency.

The **returns** and **fields** declarations and the **uses** dependency are inherited by extensions of a processor, so a derived processor needs to declare the new fields and dependencies only. 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 Section 3.

6 Implementation and Experience

Zodiac is currently implemented as a language extension to MzScheme [10], a version of Scheme [3] extended with a Java-like object system.

Zodiac has been used to implement DrScheme, a new Scheme programming environment [9]. DrScheme is a pedagogically-motivated system that helps beginners by presenting Scheme as a succession of increasingly complex languages. It also supports several extensible tools for each of these languages, such as a syntax checker, a program analyzer, *etc.* The languages are implemented as extensible datatypes in Zodiac; each tool can cope with every layer in the tower of languages.

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. Zodiac was a significant component in this rapid development. It greatly simplified the specification of the language tower, which, in turn, avoided many clerical errors and greatly 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 the protocol is low enough to be practical for such use.

Zodiac is also being applied in other domains. We have used it to build a general-purpose, extensible document construction system. This system generates "real-world" documents, and easily meets demanding performance criteria.

7 Background and Related Work

Many people, including Martin [17], Palsberg and Jay [21] and Rémy [23], have observed the trade-offs between the functional and object-oriented design approaches. They note the relative strengths and weaknesses at datatype and toolkit extension. Rémy does not offer a solution that synthesizes the respective strengths.

The literature on design patterns contains many attempts to define Interpreter- and Visitor-like patterns. The primary presentation of the Visitor pattern [11] 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. Martin [17] suggests a "sparse" version of Visitor where not every variant needs to be processed by every processor. However, he too does not address the problems of extension or of multiple tools. His solution also relies on an extensive use of dynamic casts across the class hierarchy, which reduces reliability of the program by circumventing the type system.

The most closely related work is a novel proposal by Jay and Palsberg [21]. After noting the problem of extensibility, they suggest the use of reflection to implement a Visitor-like protocol. In their protocol, all visitors are sub-classes of the Walkabout class, which provides a default visitor. The default visitor examines the argument; if it is not a base class, its fields are obtained using reflection, and each field is recursively visited. While their approach scales to classes that do not have an explicit method for the visitor, 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 relies on the ability to check whether objects are instance of a given class, which cannot be checked in all languages. They also require the implementation language to have reflective operators. Finally, but most importantly, their system is over two orders of magnitude slower than a corresponding Visitor, making it unsuitable for practical use. In contrast, our protocol works with generic object-oriented languages, and incurs almost no overhead beyond that of a Visitor.

We can alternatively view the variants of a datatype as specifying the terms in a language. Then, processors are analogous to tools like interpreters. The functional language community has been interested in the problem of creating interpreters by composing fragments that understand portions of the language [2, 7, 16, 26]. These approaches are orthogonal to ours in that they can handle semantic extensions to the interpreters, but they do not address the problem of extending the toolkit. Duggan and Sourelis [5], Findler [8] and Liang, Hudak and Jones [16] describe methods for creating extensible datatypes. However, none of these is truly 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 additions require access to the source code.

Baumgartner, Läufer and Russo [1] describe language features that facilitate the implementation of certain patterns. In contrast, a Zodiac program specifies the essence of a pattern instance. It makes explicit information that is implicit in a direct implementation of the protocol. Explicating such information provides error messages directly at the level of the pattern, rather than its implementation. It also exposes opportunities for proving theorems about the protocol and for optimizing its implementation.

8 Conclusions and Future Work

We have presented a programming protocol 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, but tools are still organized in a functional fashion, enabling extensibility in the functional dimension. Systems that implement the protocol can be extended without modification to existing code or recompilation (which is an increasingly important concern).

We have also illustrated Zodiac, a meta-language for writing extensible programs. Zodiac manages the mundane (and potentially error-prone) administrative tasks of the protocol. A variant of Zodiac has been in use for about two years in our programming environment DrScheme [9]. 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 is currently being used to build other tools, 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 our protocol incurs an execution-time 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 [22] reports that his extensible interpreter 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. A future version of Zodiac may also be able to infer some of the information that is explicitly specified in the current version.

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

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

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

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

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

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] Martin, R. C. Acyclic Visitor. In *Proceedings of the Third Annual Conference on Pattern Languages of Programs*, 1996.

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

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

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

[21] Palsberg, J. and C. B. Jay. The essence of the Visitor pattern. Technical Report 05, University of Technology, Sydney, 1997. Submitted for publication to FASE '98.

[22] Portner, N. Flexible command interpreter: A pattern for an extensible and language-independent interpreter system, 1995. Appears in [4].

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

[24] 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.

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

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

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