Design Patterns as Higher-Order Datatype-Generic Programs

Jeremy Gibbons
Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford OX1 3QD, United Kingdom
http://www.comlab.ox.ac.uk/jeremy.gibbons/

ABSTRACT

Design patterns are reusable abstractions in object-oriented software. However, using current programming languages, these elements can only be expressed extra-linguistically: as prose, pictures, and prototypes. We believe that this is not inherent in the patterns themselves, but evidence of a lack of expressivity in the languages of today. We expect that, in the languages of the future, design patterns will be expressible as reusable library code. Indeed, we claim that the languages of tomorrow will suffice; the future is not far away. The necessary features are *higher-order* and *datatype-generic* constructs; these features are already or nearly available now. We argue the case by presenting higher-order datatype-generic programs capturing ORIGAMI, a small pattern language of recursive data structures.

Categories and Subject Descriptors

F.3.3 [**Logics and meanings of programs**]: Studies of program constructs—*object-oriented constructs*; D.3.3 [**Programming languages**]: Language constructs and features—*Patterns, polymorphism, control structures, recursion*; D.3.2 [**Programming languages**]: Language classifications—*Functional languages, design languages, object-oriented languages*

General Terms

Languages, Design, Algorithms, Theory

Keywords

Design patterns, generic programming, higher-order functions, functional programming, folds, unfolds

1. INTRODUCTION

Design patterns, as the subtitle of the seminal book [17] has it, are 'elements of reusable object-oriented software'. However, within the confines of existing programming languages, these supposedly reusable elements can only be expressed extra-linguistically: as prose, pictures, and prototypes. We believe that this is not inherent in the patterns themselves, but evidence of a lack of expressivity in the languages of today. We expect that, in the languages of the future, design patterns will be expressible as directly-reusable library code. The benefits will be considerable: patterns may then be reasoned about, type-checked, applied and reused, just as any other abstractions can.

Indeed, we claim that the languages of tomorrow will suffice; the future is not far away. The necessary constructs are *higher-order* (parametrization by code) and *datatype-generic* (parametrization by type constructors) features. Higher-order constructs have been available for decades in functional programming languages such as ML [40] and Haskell [47]. Datatype genericity can be simulated in existing programming languages, but we already have significant experience and robust prototypes of languages that support it natively.

We argue our case by capturing as higher-order datatype-generic programs a small sublanguage ORIGAMI of the Gang of Four patterns. These programs are parametrized along three dimensions: by the *shape* of the computation, which is determined by the shape of the underlying data, and represented by a type constructor (an operation on types); by the *element type* (a type); and by the *body* of the computation, which is a higher-order argument (a value, typically a function).

Although our presentation is in a functional programming style, we do not intend to argue that functional programming is the paradigm of the future (whatever we might feel personally!). Rather, we believe that functional programming languages are a suitable test-bed for experimental language features — as evidenced by parametric polymorphism and list comprehensions, for example, which are both now finding their way into mainstream programming languages such as Java and C#. We expect that the evolution of programming languages will continue to follow the same trend: experimental language features will be developed and explored in small, nimble laboratory languages, and the successful experiments will eventually make their way into the outside world. Specifically, we expect that the mainstream languages of tomorrow will be broadly similar to the languages of today — strongly and statically typed, object-oriented, with an underlying imperative mindset — but incorporating additional features from the functional world — specifically, higher-order operators and datatype genericity.

2. FUNCTIONAL PROGRAMMING

We start with a brief review of the kinds of parametrization available in functional programming languages. We do this in order to

Submitted for publication, March 17, 2006

emphasize what is different about datatype-genericity, which we believe will be the next step in parametrization.

2.1 First-order, monomorphic

Functional programming is a matter of programming with expressions rather than statements, manipulating values rather than actions. For example, consider the following two datatype definitions, of lists of integers and lists of characters respectively.

```
data ListI  = NilI  | ConsI Integer ListI
data ListC = NilC | ConsC Char ListC
```

Unlike in conventional imperative (and object-oriented) programming, computations do not proceed by executing actions that destructively update a state; instead, they construct values by evaluating expressions. Thus, instead of a loop repeatedly updating a running total, summing a list of integers entails the recursive evaluation of one lengthy expression.

```
sumI  :: ListI → Integer
sumI NilI          = 0
sumI (ConsI x xs) = x + sumI xs
```

Similarly, programs to append two lists (of integers, or of characters) proceed non-destructively by recursion.

```
appendI :: ListI → ListI → ListI
appendI NilI            ys = ys
appendI (ConsI x xs)   ys = ConsI x (appendI xs ys)
appendC :: ListC → ListC → ListC
appendC NilC            ys = ys
appendC (ConsC x xs) ys = ConsC x (appendC xs ys)
```

2.2 First-order, polymorphic

The attentive reader will note that the definitions above of *appendI* and *appendC* are identical, except for their types: their monomorphic types are over-specific. Abstracting from their differences allows us to capture their commonalities. The kind of abstraction that is required is parametrization by type. This can be done both in the programs and in the datatypes they manipulate, yielding *parametrically polymorphic* datatypes and functions.

```
data List a = Nil | Cons a (List a)
sum :: List Integer → Integer
sum Nil           = 0
sum (Cons x xs) = x + sum xs
append  :: List a → List a → List a
append Nil           ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Here, for later reference, is another example of a parametrically polymorphic function.

```
concat :: List (List a) → List a
concat Nil               = Nil
concat (Cons xs xss) = append xs (concat xss)
```

2.3 Higher-order, list-specific

Each of the three programs above traverses its list argument in exactly the same way. Abstracting from their differences allows us to capture a second kind of commonality, namely the pattern of recursion. The kind of parametrization that is required is parametrization by a program; doing so yields *higher-order* programs. The common pattern is called a 'fold':

```
foldL :: b → (a → b → b) → List a → b
foldL n c Nil             = n
foldL n c (Cons x xs) = c x (foldL n c xs)
```

Instances of *foldL* replace the list constructors *Nil* and *Cons* with supplied arguments:

```
sum             = foldL 0 (+)
append xs ys = foldL ys Cons xs
concat         = foldL Nil append
```

2.4 Higher-order, tree-specific

Now, suppose one also had a polymorphic datatype of binary trees:

```
data Btree a = Tip a | Bin (Btree a) (Btree a)
```

A similar process would lead one to abstract the natural pattern of computation on these trees as another higher-order function:

```
foldB :: (a → b) → (b → b → b) → Btree a → b
foldB t b (Tip x)         = t x
foldB t b (Bin xs ys) = b (foldB t b xs) (foldB t b ys)
```

For example, instances of *foldB* reflect a tree, and flatten it to a list, in both cases replacing the tree constructors *Tip* and *Bin* with supplied constructors:

```
reverse :: Btree a → Btree a
reverse = foldB Tip nib
  where nib xs ys = Bin ys xs
flatten :: Btree a → List a
flatten = foldB wrap append
  where wrap x = Cons x Nil
```

2.5 Datatype-generic

We have seen that each kind of parametrization allows some recurring patterns to be captured. Parametric polymorphism unifies commonality of computation, abstracting from variability in irrelevant typing information. Higher-order functions unify commonality of program shape, abstracting from variability in some of the details.

But what about the two higher-order, polymorphic programs *foldL* and *foldB*? We can see that they have something in common: both replace constructors by supplied arguments; both have patterns of recursion that follow the datatype definition, with one clause per datatype variant and one recursive call per substructure. But neither parametric polymorphism nor higher-order functions suffice to capture this kind of commonality.

In fact, what differs between the two fold operators is the *shape* of the data on which they operate, and hence the shape of the programs themselves. The kind of parametrization required is by this shape; that is, by the datatype or type constructor (such as *List* or *Tree*) concerned. We call this *datatype genericity*; it allows the capture of recurring patterns in *programs of different shapes*. In Section 4 below, we explain the definition of a datatype-generic operation *fold* with the following type:

$$fold :: Bifunctor\ s \Rightarrow (s\ a\ b \rightarrow b) \rightarrow Fix\ s\ a \rightarrow b$$

Here, in addition to the type a of collection elements and the fold body (a function of type $s\ a\ b \rightarrow b$), the shape parameter s varies; the type class *Bifunctor* expresses the constraints we place on its choice. The shape parameter determines the shape of the input data; for one instantiation of s, the type $Fix\ s\ a$ is isomorphic to $List\ a$, and for another instantiation it is isomorphic to $Tree\ a$. The same shape parameter also determines the type of the fold body, supplied as an argument with which to replace the constructors.

3. GENERA OF GENERICITY

The term *generic programming* means different things to different people: to some, it means parametric polymorphism; to some, it means libraries of algorithms and data structures; to some, it means reflection and meta-programming; to some, it means polytypism. By and large, though, everyone agrees with the intention, characterized by Tim Sheard [3], of 'making programs more adaptable by making them more general'.

Generic programming usually manifests itself as a kind of parametrization. By abstracting from the differences in what would otherwise be separate but similar specific programs, one can make a single unified generic program. Instantiating the parameter in various ways retrieves the various specific programs, and ideally some new ones too. The different interpretations of the term 'generic programming' arise from different notions of what constitutes a 'parameter'.

Moreover, a parametrization is usually only called 'generic' programming if it is of a 'non-traditional' kind; by definition, traditional kinds of parametrization give rise only to traditional programming, not generic programming. Therefore, 'genericity' is in the eye of the beholder, with beholders from different programming traditions having different interpretations of the term. No doubt by-value and by-name parameter-passing mechanisms for arguments to procedures, as found in Pascal, look like 'generic programming' to an assembly-language programmer.

3.1 Parametric polymorphism

One interpretation of the term 'generic programming' is as embodied by Ada generics [41]. These were inspired by Liskov's CLU language [37], and were in turn the inspiration for the C++ template mechanism, which we discuss below. Ada *generic units* encompass 'subprograms' (functions or procedures) and 'packages' (modules or abstract datatypes). These generic units can be parametrized in various ways: by types, by values, by subprograms, and by packages. As with C++, Ada generic units are templates for their non-generic counterparts; they cannot be used until they are instantiated.

Ada generics implement a form of *parametric polymorphism*: the same generic definition is applicable to all possible instantiations. This is true for the other available kinds of formal parameter, as well as for type parameters (although Cardelli and Wegner [8] would call parametrization by these other kinds of parameter *universal polymorphism*, restricting *parametric polymorphism* to mean 'universal polymorphism with a type parameter'). For example, a value parameter could be used to parametrize a collection type by a size bound; a subprogram parameter could be used to parametrize a sorting subprogram with an ordering; a package parameter could be used to parametrize Horner's rule for polynomial evaluation by a semiring [9], an extremal path finder by a regular algebra [4, 5], or a greedy algorithm by a matroid or greedoid structure [15, 36].

3.2 Generic programming and the STL

The most popular interpretation of the term 'generic programming' is as embodied by the C++ Standard Template Library. This is an object-oriented class library providing *containers*, *iterators* and *algorithms* for many datatypes. As the name suggests, it is implemented using the C++ template mechanism, which offers similar facilities to Ada generics: class- and function templates are parametrized by type- and value parameters. (Indeed, a predecessor to the STL was an Ada library for list processing [42].) Within this community more than any other, it is considered essential that genericity imposes no performance penalty [11, 54].

The containers in the STL are parametrically polymorphic datatypes, parametrized by the element type; these are further classified into *sequence containers* (such as *Vector*, *String* and *Deque*) and *associative containers* (such as *Set*, *Multiset* and *Map*).

Bulk access to the elements of a container type is provided by *iterators*. These are abstractions of C++ pointers to elements, and so support pointer arithmetic. They are further classified according to what pointer operations they support: *input iterators* (which can only be read from, that is, dereferenceable as r-values), *output iterators* (which can only be written to, that is, dereferenceable as l-values), *forwards iterators* (which can be advanced, that is, supporting increment), *backwards iterators* (which can also retreat, that is, supporting decrement), and *random access iterators* (which can move any number of steps in one operation, that is, supporting addition).

Iterators form the interface between container types and *algorithms* over data structures. These include many general-purpose operations such as searching, sorting, filtering, and so on. Rather than operating directly on a container, an algorithm takes one or more iterators as parameters; but then the algorithm is generic, in the sense that it applies to any container that supports the appropriate kind of iterator.

In the C++ approach, the exact set of requirements on parameters (such as the iterator passed to a generic algorithm, or the element type passed to a generic container, or indeed any other template parameter) is called a *concept*. It might specify the operations available on a type parameter, the laws these operations satisfy, and the asymptotic complexities of the operations in terms of time and space; the first of these can be checked at instantiation time, but the other two cannot. The C++ template mechanism provides no means explicitly to define a concept; it is merely an informal artifact rather than a formal construct. (However, work is proceeding to formalize concepts; see for example [28, 13].)

The STL is perhaps the best known instantiation of generic programming. Indeed, some writers have taken the STL style as the definition of generic programming; for example, Siek et al [49] define generic programming as 'a methodology for program design and implementation that separates data structures and algorithms through the use of abstract requirement specifications'. We feel that this is squandering a useful term on a well-established existing practice, namely good old-fashioned abstraction.

3.3 Metaprogramming

Another interpretation of the term 'generic programming' covers various flavours of *metaprogramming*, that is, the construction of programs that write or manipulate other programs. This field encompasses *program generators* such as lex and yacc, *reflection techniques* allowing a program (typically in a dynamically-typed language) to observe and possibly modify its structure and behaviour, *generative programming* for the automated customization, configuration and assembly of components [10], and *multi-stage programming* for partitioning computations into phases [51].

In fact, the C++ template mechanism is surprisingly expressive, and already provides some kind of metaprogramming facility. Template instantiation takes place at compile time, so one can think of a C++ program with templates as a two-stage computation; several high-performance numerical libraries rely on templates' generative properties [54]. The template instantiation mechanism turns out to be Turing complete; Unruh [52] demonstrated the disquieting possibility of a program whose compilation yields the prime numbers as error messages, Czarnecki and Eisenecker [10] show the Turing-completeness of the template mechanism by implementing a rudimentary Lisp interpreter as a template meta-program, and Alexandrescu [2] presents a tour-de-force of unexpected applications of templates.

3.4 Datatype genericity

The *Datatype-Generic Programming* project at Oxford and Nottingham [23] has yet another interpretation of the term 'generic programming'. As the name suggests, *datatype-generic* programs are programs parametrized by a datatype or type functor. We motivated above the type declaration of the following datatype-generic 'fold' on multiple datatypes:

$$\begin{array}{l} \mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to \mathit{Fix}\ s\ a \to b \\ \mathit{fold}\ f = f \cdot \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \cdot \mathit{out} \end{array}$$

We explain the definition of this *fold* in Section 4; we stress that the function is parametrized not only by the type a of the elements of the datatype and the body f of the fold, but also by the shape s of the datatype itself.

One approach to datatype genericity is what is variously called *polytypism* [32], *structural polymorphism* [48] or *typecase* [56, 14], in which functions are defined inductively by case analysis on the structure of datatypes. This is the kind of genericity provided by Generic Haskell [30]. For example, here is a datatype-generic definition of encoding to a list of bits.

$$\begin{array}{ll} \mathbf{type}\ \mathit{Encode}\{[*]\} & t = t \to [\mathit{Bool}] \\ \mathbf{type}\ \mathit{Encode}\{[k \to l]\} & t = \forall a. \mathit{Encode}\{[k]\}\ a \to \\ & \qquad \mathit{Encode}\{[l]\}\ (t\ a) \end{array}$$

$$\begin{array}{ll} \mathit{encode}\{|t :: k|\} & :: \mathit{Encode}\{[k]\}\ t \\ \mathit{encode}\{|\mathit{Char}|\}\ c & = \mathit{encodeChar}\ c \\ \mathit{encode}\{|\mathit{Int}|\}\ n & = \mathit{encodeInt}\ n \\ \mathit{encode}\{|\mathit{Unit}|\}\ \mathit{unit} & = [\,] \\ \mathit{encode}\{|:+:|\}\ \mathit{ena}\ \mathit{enb}\ (\mathit{Inl}\ a) & = \mathit{False} : \mathit{ena}\ a \\ \mathit{encode}\{|:+:|\}\ \mathit{ena}\ \mathit{enb}\ (\mathit{Inr}\ b) & = \mathit{True} : \mathit{enb}\ b \\ \mathit{encode}\{|:*:|\}\ \mathit{ena}\ \mathit{enb}\ (a :*: b) & = \mathit{ena}\ a +\!\!+ \mathit{enb}\ b \end{array}$$

The generic function *encode* works for any type constructed from characters and integers using sums and products; the cases for type abstraction, application and recursion are generated automatically. Note that *encode* does not have a constant type, but a parametrized one; *type-indexed values have kind-indexed types* [29].

Because a structurally polymorphic definition is given by case analysis in this fashion, it is an *ad-hoc* form of datatype genericity. One has the flexibility to define different behaviour in different branches, and maybe even to customize the behaviour for specific types; but consequently, there is no guarantee or check that the behaviours in different branches conform, except by type. In contrast, the definition of *fold* cited above and explained below uses a *parametric* form of datatype genericity; one has less flexibility, but instances at different types necessarily behave uniformly. Ad-hoc datatype genericity is more general than parametric; for example, it is difficult to see how to define datatype-generic encoding parametrically, and conversely, any parametric definition can be expanded into an ad-hoc one. However, parametric datatype genericity offers better prospects for reasoning, and is to be preferred when it is applicable. We consider parametric datatype genericity to be the 'gold standard', and in the remainder of this paper, we concentrate on parametric datatype-generic definitions where possible.

Datatype genericity is different from the other three interpretations of generic programming outlined above. It is not just a matter of parametric polymorphism, at least not in a straighforward way; for example, parametric polymorphism abstracts from the occurrence of 'integer' in 'lists of integers', whereas datatype genericity abstracts from the occurrence of 'list'. It is not just interface conformance, as with concept satisfaction in the STL; although the latter allows *abstraction from* the shape of data, it does not allow *exploitation of* the shape of data, as required for the data compression and marshalling examples above. Finally, it is not metaprogramming: although some flavours of metaprogramming (such as reflection) can simulate datatype-generic computations, they typically do so at the cost of static checking.

4. ORIGAMI PROGRAMMING

There is a branch of the mathematics of program construction devoted to the relationship between the structure of programs and the structure of the data they manipulate [38, 39, 6, 19]. We saw a glimpse of this field in Sections 2.3 and 2.4, with the definitions of *foldL* and *foldB* respectively: the structure of each program reflects that of the datatype it traverses, for example in the number of clauses and the number and position of recursive references. In this section, we explore a little further. Folds are not the only program structure that reflects data structure, although they are often given unfair emphasis [25]; we outline *unfolds* and *builds* too, which are two kinds of dual (producing structured data rather than consuming it), and *maps*, which are special cases of these operators, and some simple combinations of these. The beauty of all of these patterns of computation is the direct relationship between their shape and that of the data they manipulate; we go on to explain how both can be parametrized by that shape, yielding *datatype-generic* patterns of computation. Elsewhere, we have called this approach *origami programming* [20].

4.1 Maps on lists

Here is the datatype of lists again.

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

The 'map' operator for a datatype applies a given function to every element of a data structure. The (higher-order, polymorphic, but list-specific) map operator for lists is given by:

$$\begin{aligned}
&mapL :: (a \to b) \to List\ a \to List\ b\\
&mapL\ f\ Nil \qquad\quad = Nil\\
&mapL\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (mapL\ f\ xs)
\end{aligned}$$

4.2 Folds on lists

The 'fold' operator for a datatype collapses a data structure down to a value. Here is the (again higher-order, polymorphic, but list-specific) fold operator for lists.

$$\begin{aligned}
&foldL :: b \to (a \to b \to b) \to List\ a \to b\\
&foldL\ e\ f\ Nil \qquad\quad = e\\
&foldL\ e\ f\ (Cons\ x\ xs) = f\ x\ (foldL\ e\ f\ xs)
\end{aligned}$$

For example, the function *filterL* (itself higher-order, polymorphic, but list-specific) takes a predicate p and a list xs, and returns the sublist of xs consisting of those elements that satisfy p.

$$\begin{aligned}
&filterL :: (a \to Bool) \to List\ a \to List\ a\\
&filterL\ p = foldL\ Nil\ (add\ p)\\
&\quad \mathbf{where}\ add\ p\ x\ xs = \mathbf{if}\ p\ x\ \mathbf{then}\ Cons\ x\ xs\ \mathbf{else}\ xs
\end{aligned}$$

As we saw in Section 2.3, the functions *sum*, *append* and *concat* are also instances of *foldL*.

4.3 Unfolds on lists

The 'unfold' operator for a datatype grows a data structure from a value. In a precise technical sense, it is the dual of the 'fold' operator. That duality isn't so obvious in the implementation for lists below, but it will become clearer with the datatype-generic version we present later.

$$\begin{aligned}
&unfoldL :: \ (b \to Bool) \to (b \to a) \to (b \to b) \to\\
&\qquad\qquad\ b \to List\ a\\
&unfoldL\ p\ f\ g\ x\\
&\quad = \mathbf{if}\ p\ x\ \mathbf{then}\ Nil\\
&\qquad\qquad \mathbf{else}\ \ Cons\ (f\ x)\ (unfoldL\ p\ f\ g\ (g\ x))
\end{aligned}$$

For example, here are two instances. The function *preds* returns the list of predecessors of a (presumed non-negative) integer; the function *takeWhile* takes a predicate p and a list xs, and returns the longest initial segment of xs all of whose elements satisfy p.

$$\begin{aligned}
&preds :: Integer \to List\ Integer\\
&preds = unfoldL\ (0 \equiv)\ id\ pred\ \mathbf{where}\ pred\ n = n-1\\
&takeWhile :: (a \to Bool) \to List\ a \to List\ a\\
&takeWhile\ p = unfoldL\ (firstNot\ p)\ head\ tail\\
&\quad \mathbf{where}\ firstNot\ p\ Nil \qquad\quad = True\\
&\qquad\qquad\ firstNot\ p\ (Cons\ x\ xs) = not\ (p\ x)
\end{aligned}$$

4.4 Origami for binary trees

We might go through a similar exercise for a datatype of internally-labelled binary trees.

$$\mathbf{data}\ Tree\ a = Empty \mid Node\ a\ (Tree\ a)\ (Tree\ a)$$

The 'map' operator applies a given function to every element of a tree.

$$\begin{aligned}
&mapT :: (a \to b) \to Tree\ a \to Tree\ b\\
&mapT\ f\ Empty \qquad\quad = Empty\\
&mapT\ f\ (Node\ x\ xs\ ys) = Node\ (f\ x)\ (mapT\ f\ xs)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (mapT\ f\ ys)
\end{aligned}$$

The 'fold' operator collapses a tree down to a value.

$$\begin{aligned}
&foldT :: b \to (a \to b \to b \to b) \to Tree\ a \to b\\
&foldT\ e\ f\ Empty \qquad\quad = e\\
&foldT\ e\ f\ (Node\ x\ xs\ ys) = f\ x\ (foldT\ e\ f\ xs)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (foldT\ e\ f\ ys)
\end{aligned}$$

For example, the function *inorder* collapses a tree down to a list.

$$\begin{aligned}
&inorder :: Tree\ a \to List\ a\\
&inorder = foldT\ Nil\ glue\\
&glue\ x\ xs\ ys = append\ xs\ (Cons\ x\ ys)
\end{aligned}$$

The 'unfold' operator grows a tree from a value.

$$\begin{aligned}
&unfoldT :: \ (b{\to}Bool) \to (b{\to}a) \to (b{\to}b) \to (b{\to}b) \to\\
&\qquad\qquad\ b \to Tree\ a\\
&unfoldT\ p\ f\ g\ h\ x\\
&\quad = \mathbf{if}\ p\ x\ \mathbf{then}\ Empty\\
&\qquad\qquad \mathbf{else}\ \ Node\ (f\ x)\ (unfoldT\ p\ f\ g\ h\ (g\ x))\\
&\qquad\qquad\qquad\qquad\qquad\ (unfoldT\ p\ f\ g\ h\ (h\ x))
\end{aligned}$$

For example, the Calkin–Wilf tree contains each of the positive rationals exactly once [1, 26]:

$$\begin{aligned}
&cwTree :: Tree\ Rational\\
&cwTree = unfoldT\ (const\ False)\ frac\ left\ right\ (1,1)\\
&\quad \mathbf{where}\ frac\ (m,n) \ \ = m \,\%\, n\\
&\qquad\qquad\ left\ (m,n) \ \ = (m, m+n)\\
&\qquad\qquad\ right\ (m,n) = (n+m, n)
\end{aligned}$$

(Here, *const a* is the function that always returns a, and the operator % constructs a rational from its numerator and denominator.)

4.5 Aside: ad-hoc polymorphism

For another example of an unfold to trees, consider the function *grow* that generates a binary search tree from a list of elements whose type supports an ordering.

$$\begin{aligned}
&grow :: Ord\ a \Rightarrow List\ a \to Tree\ a\\
&grow = unfoldT\ isNil\ head\ littles\ bigs\\
&littles\ (Cons\ x\ xs) = filterL\ (\leqslant x)\ xs\\
&bigs\ (Cons\ x\ xs) \ \ = filterL\ (not \cdot (\leqslant x))\ xs
\end{aligned}$$

The '$Ord\ a \Rightarrow$' is a Haskell *type class context*, providing *ad-hoc polymorphism*. This is not parametric polymorphism, because the $\leqslant$ operator (and hence also *grow*) is not defined for all element types a; moreover, for those types on which $\leqslant$ is defined, it is defined in quite different ways. The type class *Ord* might be defined in Haskell as follows:

$$\begin{aligned}
&\mathbf{class}\ Ord\ a\ \mathbf{where}\\
&\quad (\leqslant) :: a \to a \to Bool
\end{aligned}$$

(In fact, the definition is more complex than this; but this will serve for illustration.) Various types are instance of the type class, by virtue of supporting a comparison operation:

$$\begin{aligned}
&\mathbf{instance}\ Ord\ Integer\ \mathbf{where}\\
&\quad (m \leqslant n) = isNonNegative\ (n-m)
\end{aligned}$$

We explain this mechanism here, because we will use it again later.

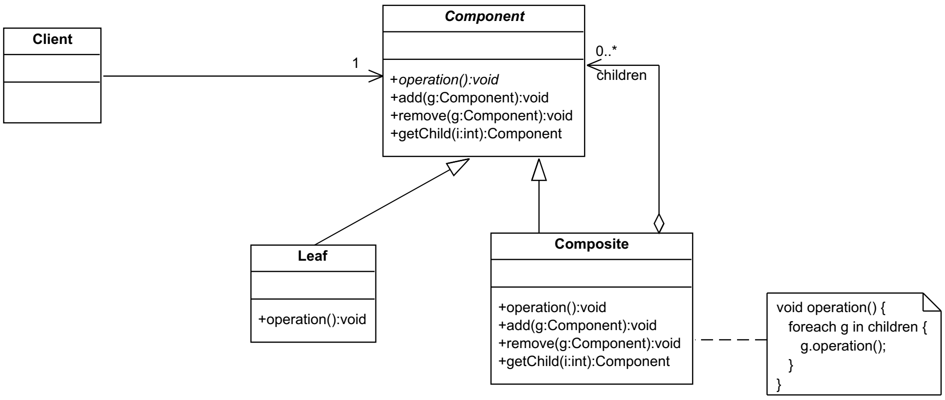

Figure 1: The class structure of the COMPOSITE pattern

4.6 Hylomorphisms

An unfold followed by a fold is a common pattern of computation [39]; the unfold generates a data structure, and the fold immediately consumes it. For example, here is a (higher-order, polymorphic, but list-specific) hylomorphism operator for lists, and an instance for computing factorials: first generate the predecessors of the input using an unfold, then compute the product of these predecessors using a fold.

$$\begin{aligned}&\mathit{hyloL} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to \\ &\qquad c \to (a \to c \to c) \to b \to c \\ &\mathit{hyloL}\ p\ f\ g\ e\ h = \mathit{foldL}\ e\ h \cdot \mathit{unfoldL}\ p\ f\ g\end{aligned}$$

$$\begin{aligned}&\mathit{fact} :: \mathit{Integer} \to \mathit{Integer} \\ &\mathit{fact} = \mathit{hyloL}\ (0\ \equiv)\ \mathit{id}\ \mathit{pred}\ 1\ (*)\end{aligned}$$

With lazy evaluation, the intermediate data structure is not computed all at once. It is produced on demand, and each demanded cell consumed immediately. In fact, the intermediary can be *deforested* altogether.

$$\begin{aligned}&\mathit{hyloL} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to \\ &\qquad c \to (a \to c \to c) \to b \to c \\ &\mathit{hyloL}\ p\ f\ g\ e\ h\ x \\ &\quad = \mathbf{if}\ p\ x\ \mathbf{then}\ e \\ &\qquad\quad \mathbf{else}\ \ h\ (f\ x)\ (\mathit{hyloL}\ p\ f\ g\ e\ h\ (g\ x))\end{aligned}$$

A similar definition can be given for binary trees, as shown below, together with an instance giving a kind of quicksort.

$$\begin{aligned}&\mathit{hyloT} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to (b \to b) \to \\ &\qquad c \to (a \to c \to c \to c) \to b \to c \\ &\mathit{hyloT}\ p\ f\ g1\ g2\ e\ h\ x \\ &\quad = \mathbf{if}\ p\ x\ \mathbf{then}\ e \\ &\qquad\quad \mathbf{else}\ \ h\ (f\ x)\ (\mathit{hyloT}\ p\ f\ g1\ g2\ e\ h\ (g1\ x)) \\ &\qquad\qquad\qquad\quad (\mathit{hyloT}\ p\ f\ g1\ g2\ e\ h\ (g2\ x))\end{aligned}$$

$$\begin{aligned}&\mathit{qsort} :: \mathit{Ord}\ a \Rightarrow \mathit{List}\ a \to \mathit{List}\ a \\ &\mathit{qsort} = \mathit{hyloT}\ \mathit{isNil}\ \mathit{head}\ \mathit{littles}\ \mathit{bigs}\ \mathit{Nil}\ \mathit{glue}\end{aligned}$$

4.7 Short-cut fusion

Unfolds capture a highly structured pattern of computation for generating recursive data structures. There exist slight generalizations of unfolds, such as *monadic unfolds* [45, 46], *apomorphisms* [55] and *futumorphisms* [53], but these still all conform to the same structural scheme, and not all programs that generate data structures fit this scheme [24]. Gill et al. [27] introduced an operator they called *build* for unstructured generation of data, in order to simplify the implementation and broaden the applicability of deforestation optimizations as discussed in the previous section.

The idea behind *build* is to allow the identification of precisely where in a program the nodes of a data structure are being generated; then it is straightforward for a compiler to fuse a following fold, inlining functions to replace those constructors and deforesting the data structure altogether. The definition of *build* is reminiscent of the continuation-passing style beloved of Scheme programmers; it takes as argument a program with 'holes' for constructors, and plugs those holes with actual constructors.

$$\begin{aligned}&\mathit{buildL} :: (\forall b.\ \ b \to (a \to b \to b) \to b) \to \mathit{List}\ a \\ &\mathit{buildL}\ g = g\ \mathit{Nil}\ \mathit{Cons}\end{aligned}$$

The function *buildL* has a rank-two type; the argument g must be parametrically polymorphic in the constructor arguments, in order to ensure that all uses of the constructors are abstracted. We argued above that *unfoldL* is a dual to *foldL* in one sense; we will make that sense clear in Section 4.11. In another sense, *buildL* is *foldL*'s dual: whereas the fold deletes constructors and replaces them with something else, the build inserts those constructors.

The beauty of the idea is that fusion with a following fold is simple to state:

$$\mathit{foldL}\ e\ f\ (\mathit{buildL}\ g) = g\ e\ f$$

Perhaps more importantly, it is also easy for a compiler to exploit.

Build operators are strictly more expressive than unfolds. For instance, it is possible to define *unfoldL* in terms of *buildL*:

$$\begin{aligned}&\mathit{unfoldL} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to \\ &\qquad b \to \mathit{List}\ a \\ &\mathit{unfoldL}\ p\ f\ g\ b = \mathit{buildL}\ (h\ b)\ \mathbf{where} \\ &\quad h\ b\ n\ c = \mathbf{if}\ p\ b\ \mathbf{then}\ n\ \mathbf{else}\ c\ (f\ b)\ (h\ (g\ b)\ n\ c)\end{aligned}$$

However, some functions that generate lists can be expressed as an instance of *buildL* and not of *unfoldL*; one such example is the function that computes the infinite list of multiples of an integer [24]:

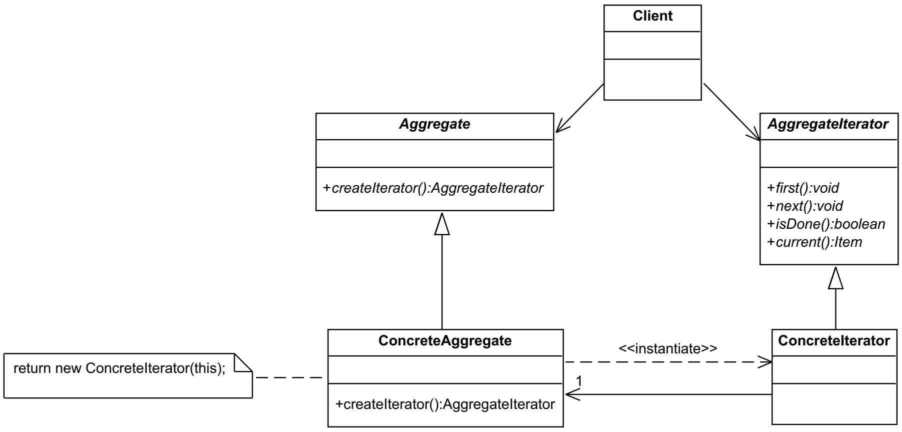

Figure 2: The class structure of the ITERATOR pattern

```
mults :: Int → List Int
mults n = buildL (next n 0) where
  next :: Int → Int → b → (Int → b → b) → b
  next i j n c = c j (next i (i+j) n c)
```

The disadvantage of *buildL* compared to *unfoldL* is a consequence of its unstructured approach: the former does not support the powerful *universal properties* that greatly simplify program calculation with the latter [19].

Of course, there is nothing special about lists in this regard. One can define build operators for any datatype:

```
buildT :: (∀b. b → (a → b → b → b) → b) → Tree a
buildT g = g Empty Node
```

4.8 Datatype genericity

As we have already argued, data structure determines program structure. It therefore makes sense to abstract from the determining shape. We do this by defining a datatype *Fix*, parametrized both by an element type *a* of basic kind (a plain type, such as integers or strings), and by a shape type *s* of higher kind (a type constructor, such as 'pairs of' or 'lists of').

```
data Fix s a = In (s a (Fix s a))
out :: Fix s a → s a (Fix s a)
out (In x) = x
```

Equivalently, we could use a record type with a single named field, and define both the constructor *In* and the destructor *out* at once.

```
data Fix s a = In{out :: s a (Fix s a)}
```

4.9 Specific datatypes

The parameter *s* determines the shape; '*Fix*' ties the recursive knot. Here are three instances of *Fix* using different shapes: lists and internally-labelled binary trees as seen before, and also externally-labelled binary trees.

```
data ListF a b = NilF | ConsF a b
type List a = Fix ListF a
data TreeF a b = EmptyF | NodeF a b b
type Tree a = Fix TreeF a
data BtreeF a b = TipF a | BinF b b
type Btree a = Fix BtreeF a
```

Note that *Fix s a* is a recursive type. Typically, as in the three instances above, the shape *s* has several variants, including a non-recursive base case. But with lazy evaluation, infinite structures are possible, and so the definition makes sense even with no base case. For example, *Fix Pair a* is a type of infinite internally-labelled binary trees (and would suffice for the *cwTree* example above).

4.10 Bifunctors

Not all valid binary type constructors *s* are suitable for *Fix*ing (for example, because of function types). It turns out that we should restrict attention to *bifunctors*, which support a *bimap* operation 'locating' all the elements. We capture this constraint as a type class.

```
class Bifunctor s where
  bimap :: (a → c) → (b → d) → s a b → s c d
```

Technically speaking, *bimap* should satisfy some properties:

```
bimap id id           = id
bimap f g · bimap h j = bimap (f · h) (g · j)
```

These cannot be expressed in Haskell — but we might expect to be able to express them in the languages of tomorrow.

All datatypes made from sum and product constructors induce bifunctors. Here are instances for our three example shapes.

```
instance Bifunctor ListF where
  bimap f g NilF         = NilF
  bimap f g (ConsF x y) = ConsF (f x) (g y)
instance Bifunctor BtreeF where
  bimap f g (TipF x)    = TipF (f x)
  bimap f g (BinF y z) = BinF (g y) (g z)
instance Bifunctor TreeF where
  bimap f g EmptyF         = EmptyF
  bimap f g (NodeF x y z) = NodeF (f x) (g y) (g z)
```

The type signature of the operator *bimap* is datatype-generic, since it is parameterized by the shape *s* of the data:

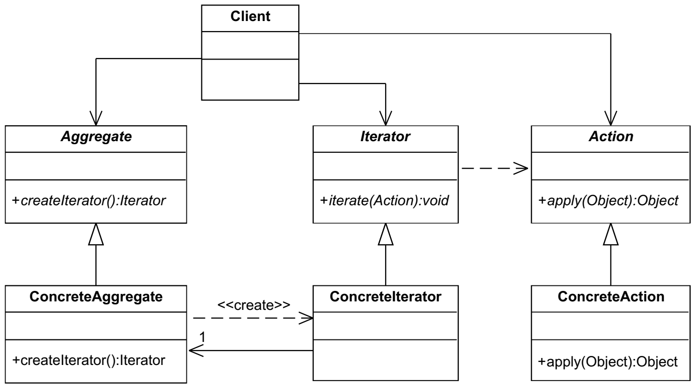

Figure 3: The class structure of an internal ITERATOR

$$
\begin{array}{l}
\mathit{bimap} :: \mathit{Bifunctor}\ s \Rightarrow \\
\quad (a \to c) \to (b \to d) \to s\ a\ b \to s\ c\ d
\end{array}
$$

However, because *bimap* is encoded as a member function of a type class, the definitions for particular shapes are examples of ad-hoc rather than parametric datatype genericity; each instance entails a proof obligation that the appropriate laws are satisfied.

It is a bit tedious to have to provide a new instance of *Bifunctor* for each new datatype shape; one would of course prefer a single datatype-generic definition. This is the kind of feature for which Generic Haskell [30] is designed, and one can almost achieve the same effect in Haskell [14]. One might hope that these instance definitions could in fact be inferred, in the languages of tomorrow. But whatever the implementation mechanism, the result will still be ad-hoc datatype-generic: it is necessarily the case that different code is used to locate the elements within data of different shapes.

4.11 Datatype-generic recursion patterns

The datatype-specific recursion patterns introduced above can be made generic in the shape *s*, provided that this is a bifunctor.

$$
\begin{array}{l}
\mathit{map} :: \mathit{Bifunctor}\ s \Rightarrow \\
\qquad (a \to b) \to \mathit{Fix}\ s\ a \to \mathit{Fix}\ s\ b \\
\mathit{map}\ f = \mathit{In} \cdot \mathit{bimap}\ f\ (\mathit{map}\ f) \cdot \mathit{out} \\
\mathit{fold} :: \quad \mathit{Bifunctor}\ s \Rightarrow \\
\qquad (s\ a\ b \to b) \to \mathit{Fix}\ s\ a \to b \\
\mathit{fold}\ f = f \cdot \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \cdot \mathit{out} \\
\mathit{unfold} :: \mathit{Bifunctor}\ s \Rightarrow \\
\qquad (b \to s\ a\ b) \to b \to \mathit{Fix}\ s\ a \\
\mathit{unfold}\ f = \mathit{In} \cdot \mathit{bimap}\ \mathit{id}\ (\mathit{unfold}\ f) \cdot f \\
\mathit{hylo} :: \quad \mathit{Bifunctor}\ s \Rightarrow \\
\qquad (b \to s\ a\ b) \to (s\ a\ c \to c) \to b \to c \\
\mathit{hylo}\ f\ g = g \cdot \mathit{bimap}\ \mathit{id}\ (\mathit{hylo}\ f\ g) \cdot f \\
\mathit{build} :: \quad \mathit{Bifunctor}\ s \Rightarrow \\
\qquad (\forall b.\ (s\ a\ b \to b) \to b) \to \mathit{Fix}\ s\ a \\
\mathit{build}\ f = f\ \mathit{In}
\end{array}
$$

The datatype-generic definitions are surprisingly short — shorter even than the datatype-specific ones. The structure becomes much clearer with the higher level of abstraction. In particular, the duality between *fold* and *unfold* is obvious.

5. ORIGAMI: A PATTERN LANGUAGE

In this section we describe ORIGAMI, a little pattern language for recursive data structures, consisting of four of the Gang of Four design patterns [17]:

- COMPOSITE, for modelling recursive structures;
- ITERATOR, for linear access to the elements of a composite;
- VISITOR, for structured traversal of a composite;
- BUILDER, to generate a composite structure.

These four patterns belong together. They all revolve around the notion of a hierarchical structure, represented as a COMPOSITE. One way of constructing such hierarchies is captured by the BUILDER pattern: a client application knows what kinds of part to add and in what order, but it delegates to a separate object knowledge of their implementation and responsibility for creating and holding them. Having constructed a hierarchy, there are two kinds of traversal we might perform over it: either considering it as a container of elements, in which case we use an ITERATOR for a linear traversal; or considering its shape as significant, in which case we use a VISITOR for a structured traversal.

5.1 Composite

The COMPOSITE pattern 'lets clients treat individual objects and compositions of objects uniformly', by 'composing objects into tree structures'. The essence of the pattern is a common supertype, of which both atomic and aggregated objects are subtypes, as shown in Figure 1.

5.2 Iterator

The ITERATOR pattern 'provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation'. It does this by separating the responsibilities of containment and iteration. The standard implementation is as an *external* or client-driven iterator, illustrated in Figure 2 and as embodied for example in the Java standard library.

In addition to the standard implementation, GOF also discuss *internal* or iterator-driven ITERATORs, illustrated in Figure 3. These might be modelled by the following pair of interfaces:

```
public interface Action { Object apply (Object o); }
public interface Iterator { void iterate (Action a); }
```

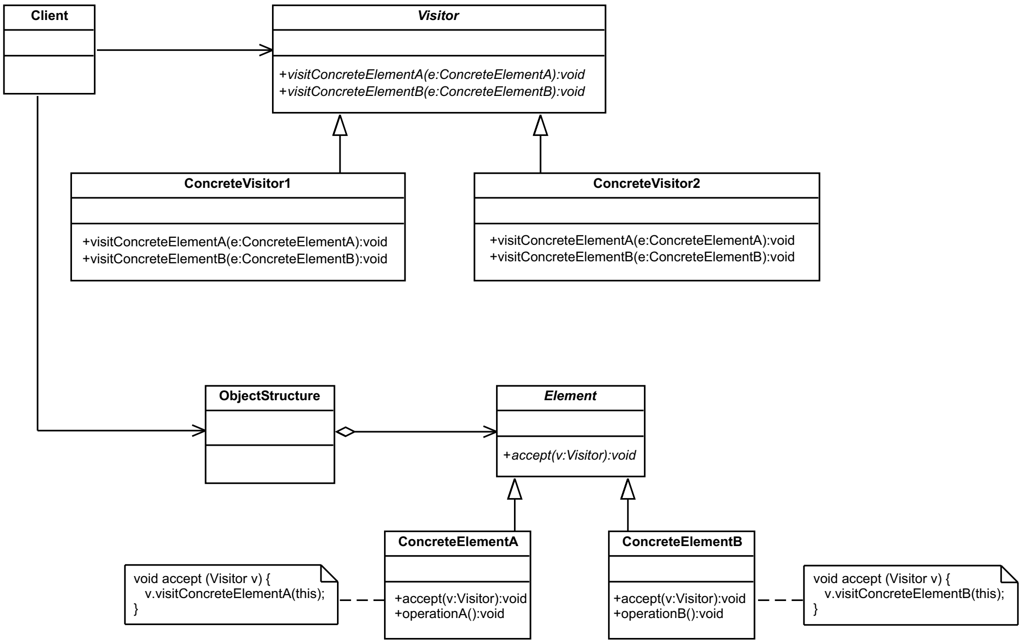

Figure 4: The class structure of the VISITOR pattern

An object implementing the *Action* interface provides a single method *apply*, which takes in a collection element and returns (either a new, or the same but modified) element. (The C++ STL calls such objects 'functors', but we avoid that term here to prevent name clashes with type functors.) A collection (implements a FACTORY METHOD to return a separate subobject that) implements the *Iterator* interface to accept an *Action*, apply it to each element in turn, and replace the original elements with the possibly new ones returned. Internal ITERATORs are less flexible than external — for example, it is more difficult to have two linked iterations over the same collection, and to terminate an iteration early — but they are correspondingly simpler to use.

5.3 Visitor

In the normal object-oriented paradigm, the definition of each traversal operation is spread across the whole class hierarchy of the structure being traversed — typically but not necessarily a COMPOSITE. This makes it easy to add new variants of the datatype (for example, new kinds of leaf node in the COMPOSITE), but hard to add new traversal operations.

The VISITOR pattern 'represents an operation to be performed on the elements of an object structure', allowing one to 'define a new operation without changing the classes of the elements on which it operates'. This is achieved by providing a hook for associating new traversals (the method *accept* in Figure 4), and an interface for those traversals to implement. This is a kind of *aspect-oriented programming* [35], modularizing what would otherwise be a cross-cutting concern. It reverses the costs: it is now easy to add new traversals, but hard to add new variants. (Phil Wadler [59] has coined the term *expression problem* for this tension between dimensions of easy extension.)

5.4 Builder

Finally, the BUILDER pattern 'separates the construction of a complex object from its representation, so that the same construction process can create different representations'. As Figure 5 shows, this is done by delegating responsibility for the construction to a separate object — in fact, a STRATEGY for performing the construction.

The GOF motivating example of the BUILDER pattern involves assembling a product that is basically a simple collection; that is necessarily the case, because the operations supported by a builder object add parts and return void. However, they also suggest the possibility of building a more structured product, in which the parts are linked together. For example, to construct a tree, each operation to add a part could return a unique identifier for the part added, and take an optional identifier for the parent to which to add it.

GOF also suggest the possibility of BUILDERs that compute. Instead of constructing a large *Product* and eventually collapsing it, one can provide a separate implementation of the *Builder* interface that makes the *Product* itself the collapsed result, computing it on the fly while building.

5.5 An example

As an example of the ORIGAMI pattern language, consider the little document system illustrated in Figure 6. (The complete code is given in an appendix, for reference.)

- The focus of the application is a COMPOSITE structure of documents: *Section*s have a *title* and a collection of sub-*Component*s, and *Paragraph*s have a *body*.
- One can iterate over such a structure using an internal ITERATOR, which acts on every *Paragraph*. For instance, iterating with a *SpellCorrector* might correct the spelling of every paragraph body. (For brevity, we have omitted the possibility of acting on the *Section*s of a document, but it would be easy to extend the *Action* interface to allow this. We have also made the *apply* method return *void*, so providing no way to

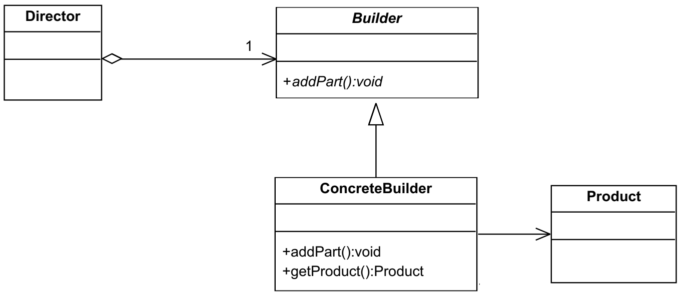

Figure 5: The class structure of the BUILDER pattern

change the identity of the document elements; more generally, *apply* could optionally return new elements.)

- One can also traverse the document structure with a VISITOR, for example to compute some summary of the document. For instance, a *PrintVisitor* might yield a string array with the section titles and paragraph bodies in order.
- Finally, one can construct such a document using a BUILDER. We have used the structured variant, adding *Section*s and *Paragraph*s as children of existing *Component*s via unique *int* identifiers. A *ComponentBuilder* constructs a *Component* as expected, whereas a *PrintBuilder* incorporates the printing behaviour of the *PrintVisitor* incrementally, actually constructing a string array instead.

This one application is a paradigmatic example of each of the four ORIGAMI patterns. We therefore claim that any alternative representation of the patterns cleanly capturing this structure is a faithful rendition of those patterns. In Section 6 below, we provide just such a representation, in terms of higher-order datatype-generic programs. Section 6.5 justifies our claim of a faithful rendition by capturing the structure of the document application in this alternative representation.

6. PATTERNS AS HODGPS

We revisit the ORIGAMI pattern language from Section 5, showing that each of the four patterns can be captured as a higher-order datatype-generic program. However, we consider them in a slightly different order; it turns out that the datatype-generic representation of the ITERATOR pattern builds on that of VISITOR.

6.1 Composite in HODGP

COMPOSITEs are just recursive data structures. So actually, these do not correspond to programs, but to types. Recursive data structures come essentially for free in functional programming languages.

$$\mathbf{data}\ \mathit{Fix}\ s\ a = \mathit{In}\{\mathit{out} :: s\ a\ (\mathit{Fix}\ s\ a)\}$$

What is datatype-generic about this definition is that it is parametrized by the shape s of the data structure; thus, one recursive datatype serves to capture *all* recursive data structures, whatever their shape.

6.2 Visitor in HODGP

The VISITOR pattern collects fragments of each traversal into one place, and provides a hook for performing such traversals. The resulting style matches the normal functional programming paradigm, in which traversals are entirely separate from the data structures traversed. No explicit hook is needed; the connection between traversal and data is made within the traversal by dispatching on the data, either by pattern matching or (equivalently) by applying a destructor. A common case of such traversals, albeit not the most general, is the fold operator introduced above.

$$\begin{aligned}&\mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow\\ &\qquad (s\ a\ b \rightarrow b) \rightarrow \mathit{Fix}\ s\ a \rightarrow b\\ &\mathit{fold}\ f = f \cdot \mathit{bimap}\ \mathit{id}\ (\mathit{fold}\ f) \cdot \mathit{out}\end{aligned}$$

This too is datatype-generic, parametrized by the shape s: the same function *fold* suffices to traverse any shape of COMPOSITE structure.

6.3 Iterator in HODGP

External ITERATORs give sequential access to the elements of collection. The functional approach would be to provide a view of the collection as a list of elements. Seen this way, the ITERATOR pattern is a special case of the VISITOR pattern, traversing using a body *combiner* that combines the element lists from substructures into one overall element list.

$$\begin{aligned}&\mathit{contents} :: \mathit{Bifunctor}\ s \Rightarrow\\ &\qquad\qquad (s\ a\ (\mathit{List}\ a) \rightarrow \mathit{List}\ a) \rightarrow \mathit{Fix}\ s\ a \rightarrow \mathit{List}\ a\\ &\mathit{contents}\ \mathit{combiner} = \mathit{fold}\ \mathit{combiner}\end{aligned}$$

With lazy evaluation, the list of elements can be generated incrementally on demand, rather than eagerly in advance: 'lazy evaluation means that lists and iterators over lists are identified' [58].

In the formulation above, the *combiner* argument has to be provided to the *contents* operation. Passing different *combiner*s allows the same COMPOSITE to yield its elements in different orders; for example, a tree-shaped container could support both preorder and postorder traversal. On the other hand, it is clumsy always to have to specify the *combiner*. One could specify it once and for all, in the class *Bifunctor*, in effect making it another datatype-generic operation parametrized by the shape s. In the languages of tomorrow, one might expect that at least one, obvious implementation of *combiner* could be inferred automatically.

Of course, some aspects of external ITERATORs can already be expressed linguistically; the interface *java.util.Iterator* has been available for years in the Java API, the iterator concept has been explicit in the C++ Standard Template Library for even longer, and recent versions of Java and C# even provide language support ('**foreach**') for iterating over the elements yielded by such an oper-

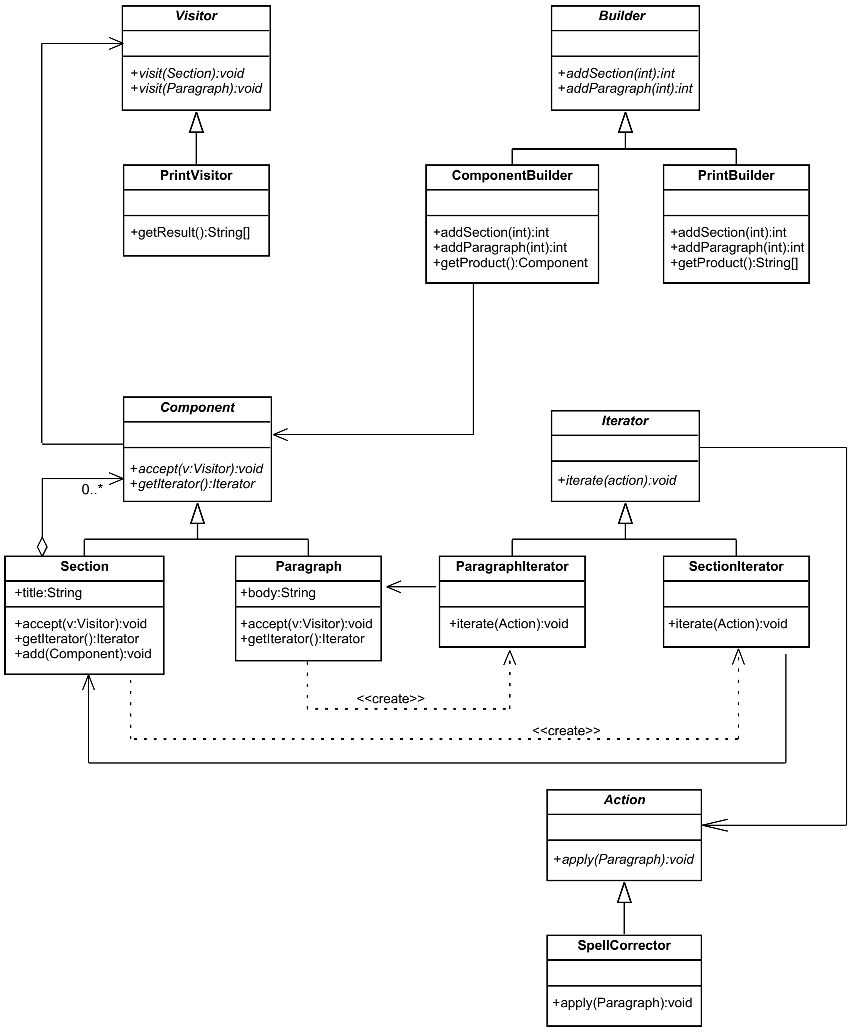

Figure 6: An application of the ORIGAMI pattern language

ator. Thus, element consumers can be written datatype-generically. But still, one has to implement the *Iterator* anew for each datatype defined; element producers are still datatype-specific.

An internal ITERATOR is basically a map operation, iterating over a collection and yielding one of the same shape but with different or modified elements. In HODGP, we can give a *single generic* definition of this.

$$
\begin{aligned}
&map :: Bifunctor\ s \Rightarrow \\
&\qquad (a \to b) \to Fix\ s\ a \to Fix\ s\ b \\
&map\ f = In \cdot bimap\ f\ (map\ f) \cdot out
\end{aligned}
$$

This is in contrast with the object-oriented approach, in which the *Iterator* interface is generic but its implementations are datatype-specific.

6.4 Builder in HODGP

The standard protocol for the BUILDER pattern involves a *Director* sending *Part*s one by one to a *Builder* for it to assemble, and then retrieving from the *Builder* a *Product*. Thus, the product is assembled in a step-by-step fashion, but is unavailable until assembly is complete. With lazy evaluation, we can in some circumstances construct the *Product* incrementally: we can yield access to the root of the product structure while continuing to assemble its substructures. In the case that the data structure is assembled in a regular fashion, this corresponds in the HODGP style to an unfold operation.

$$
\begin{aligned}
&unfold :: Bifunctor\ s \Rightarrow \\
&\qquad (b \to s\ a\ b) \to b \to Fix\ s\ a \\
&unfold\ f = In \cdot bimap\ id\ (unfold\ f) \cdot f
\end{aligned}
$$

When the data structure is assembled irregularly, a build operator has to be used instead.

$$
\begin{aligned}
&build :: Bifunctor\ s \Rightarrow \\
&\qquad (\forall b.\ (s\ a\ b \to b) \to b) \to Fix\ s\ a \\
&build\ f = f\ In
\end{aligned}
$$

These are both datatype-generic programs, parametrized by the shape of product to be built. In contrast, the GOF BUILDER pattern states the general scheme, but requires code specific for each *Builder* interface and each *ConcreteBuilder* implementation.

Turning to GOF's computing builders, with lazy evaluation there is not so pressing a need to fuse building with postprocessing. If the structure of the consumer computation matches that of the producer — in particular, if the consumer is a fold and the producer a build or an unfold — then consumption can be interleaved with production, and the whole product never need be in existence.

Nevertheless, naive interleaving of production and consumption of parts of the product still involves the creation and immediate disposal of those parts. Even the individual parts need never be constructed; often, they can be deforested [57], with the attributes of a part being fed straight into the consumption process. When the producer is an unfold, the composition of producer and consumer is (under certain mild strictness conditions) a hylomorphism.

$$
\begin{aligned}
&hylo :: Bifunctor\ s \Rightarrow \\
&\qquad (b \to s\ a\ b) \to (s\ a\ c \to c) \to b \to c \\
&hylo\ f\ g = g \cdot bimap\ id\ (hylo\ f\ g) \cdot f
\end{aligned}
$$

More generally, but less conveniently for reasoning, the producer is a build, and the composition simply replaces the constructors in the builder by the body of the fold.

$$
\begin{aligned}
&foldBuild :: Bifunctor\ s \Rightarrow \\
&\qquad (\forall b.\ (s\ a\ b \to b) \to b) \to (s\ a\ b \to b) \to b \\
&foldBuild\ f\ g = f\ g
\end{aligned}
$$

Once again, both definitions are datatype-generic; both takes as arguments a producer f and a consumer g, both with types parametrized by the shape s of the product to be built. Note especially that in both cases, the fusion requires no creativity; in contrast, GOF's computing builders can take considerable insight and ingenuity to program (as we shall see in the appendix).

6.5 The example, revisited

To justify our claim that the higher-order datatype-generic representation of the ORIGAMI pattern language is a faithful rendition, we use it to re-express the document application discussed in Section 5.5 and illustrated in Figure 6.

- The COMPOSITE structure has the following shape.

  $$
  \begin{aligned}
  &\mathbf{data}\ DocF\ a\ b = Para\ a \mid Sec\ String\ [b] \\
  &\mathbf{type}\ Doc = Fix\ DocF\ String \\
  &\mathbf{instance}\ Bifunctor\ DocF\ \mathbf{where} \\
  &\quad bimap\ f\ g\ (Para\ s) \quad = Para\ (f\ s) \\
  &\quad bimap\ f\ g\ (Sec\ s\ xs) = Sec\ s\ (map\ g\ xs)
  \end{aligned}
  $$

  We have chosen to consider paragraph bodies as the 'contents' of the data structure, but section titles as part of the 'shape'; that decision could be varied.

- We used an ITERATOR to implement the *SpellCorrector*; this would be modelled now as an instance of *map*.

  $$
  \begin{aligned}
  &correct :: String \to String \\
  &corrector :: Doc \to Doc \\
  &corrector = map\ correct
  \end{aligned}
  $$

- The use of VISITOR to print the contents of a document is a paradigmatic instance of a *fold*.

  $$
  \begin{aligned}
  &printDoc :: Doc \to [String] \\
  &printDoc = fold\ combine \\
  &combine :: DocF\ String\ [String] \to [String] \\
  &combine\ (Para\ s) \quad = [s] \\
  &combine\ (Sec\ s\ xs) = s : concat\ xs
  \end{aligned}
  $$

- Finally, in place of the BUILDER pattern, we can use *unfold* for constructing documents, at least when doing so in a structured fashion. For example, consider the following simple representation of XML documents.

  $$
  \begin{aligned}
  &\mathbf{data}\ XML = Text\ String \mid Entity\ Tag\ Attrs\ [XML] \\
  &\mathbf{type}\ Tag = String \\
  &\mathbf{type}\ Attrs = [(String, String)]
  \end{aligned}
  $$

  From such an XML document we can construct one of our documents, with *Text* elements as paragraphs and *Entity*s as sections with appropriate titles.

  $$
  \begin{aligned}
  &fromXML :: XML \to Doc \\
  &fromXML = unfold\ step \\
  &step :: XML \to DocF\ String\ XML \\
  &step\ (Text\ s) \qquad\qquad = Para\ s \\
  &step\ (Entity\ t\ kvs\ xs) = Sec\ (title\ t\ kvs)\ xs
  \end{aligned}
  $$

$$
\begin{aligned}
&\mathit{title} :: \mathit{Tag} \to \mathit{Attrs} \to \mathit{String}\\
&\mathit{title}\ t\ [\,] = t\\
&\mathit{title}\ t\ \mathit{kvs} = t +\!\!+ \mathit{paren}\ (\mathit{join}\ (\mathit{map}\ \mathit{attr}\ \mathit{kvs}))\ \mathbf{where}\\
&\quad \mathit{paren}\ s \quad\ = \texttt{" ("} +\!\!+ s +\!\!+ \texttt{")"}\\
&\quad \mathit{join}\ [s] \quad\ = s\\
&\quad \mathit{join}\ (s : \mathit{ss}) = s +\!\!+ \texttt{", "} +\!\!+ \mathit{join}\ \mathit{ss}\\
&\quad \mathit{attr}\ (k, v) \ \ = k +\!\!+ \texttt{"='"} +\!\!+ v +\!\!+ \texttt{"'"}
\end{aligned}
$$

Printing of a document constructed from an XML file is the composition of a fold with an unfold, and so a hylomorphism:

$$
\begin{aligned}
&\mathit{printXML} :: \mathit{XML} \to [\mathit{String}]\\
&\mathit{printXML} = \mathit{hylo}\ \mathit{step}\ \mathit{combine}
\end{aligned}
$$

- For constructing documents in a less structured fashion, we have to resort to the more general and more complicated *build* operator. For example, here is a builder for a simple document of one section with two sub-paragraphs.

$$
\begin{aligned}
&\mathit{buildDoc} :: (\mathit{DocF}\ \mathit{String}\ b \to b) \to b\\
&\mathit{buildDoc}\ f = f\ (\mathit{Sec}\ \texttt{"Heading"}\ [f\ (\mathit{Para}\ \texttt{"p1"}),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad f\ (\mathit{Para}\ \texttt{"p2"})])
\end{aligned}
$$

We can actually construct the document from this builder, simply by passing it to the operator *build*, which plugs the holes with document constructors.

$$
\begin{aligned}
&\mathit{myDoc} :: \mathit{Doc}\\
&\mathit{myDoc} = \mathit{build}\ \mathit{buildDoc}
\end{aligned}
$$

If we want to traverse the resulting document, for example to print it, we can do so directly without having to construct the document in the first place; we do so by plugging the holes instead with the body of the *printDoc* fold.

$$
\begin{aligned}
&\mathit{printMyDoc} :: [\mathit{String}]\\
&\mathit{printMyDoc} = \mathit{buildDoc}\ \mathit{combine}
\end{aligned}
$$

7. DISCUSSION

We have shown that two language features — *higher-order functions* and *datatype genericity* — suffice to capture as reusable code a number of the familiar GOF design patterns; specifically, the patterns we have considered are COMPOSITE, ITERATOR, VISITOR and BUILDER, which together we call the ORIGAMI pattern language.

Our intentions in doing so are not so much to criticize the existing informal presentations of these four and other patterns. Rather, we aim to promote the uptake of these advanced language techniques, and encourage their incorporation in mainstream programming languages. In this regard, we are following in the footsteps of Norvig [43], who wrote that 16 of the 23 GOF patterns are 'invisible or simple' in Lisp, and others who argue that design patterns amount to admissions of inexpressiveness in programming languages. However, in contrast to Norvig and the others favouring dynamic languages [50], our presentation provides genericity while preserving strong static typing.

We do not claim to have done away with the need for all 23 of the GOF patterns, or for that matter with any deuterocanonical ones either. In particular, we do not see yet how to capture *creational* design patterns as higher-order datatype-generic programs. This is perhaps because our approach is to model object-oriented ideas in a functional framework, and that framework has no direct analogue of object creation. However, we hope and expect that the languages of tomorrow will provide higher-order datatype-generic features in a more traditional framework, and then we may be able to make better progress. Indeed, Alexandrescu's *type lists* technique for a GENERIC ABSTRACT FACTORY [2] is essentially a datatype-generic metaprogram written with C++ templates.

We also appreciate that there is more to design patterns than their extensional characteristics, which can be expressed as class and sequence diagrams and captured as programs or programming constructs. Also important are their intensional characteristics: motivation for their use, paradigmatic examples, trade-offs in their application, and other aspects of the 'story' behind the pattern. Our presentation impinges only on the limited extensional aspects of those patterns we treat.

8. RELATED WORK

This paper is based on ideas from the Algebra of Programming ('Squiggol') community, and especially the work of Roland Backhouse and Grant Malcolm [38, 5], Richard Bird and Oege de Moor [6, 7], Maarten Fokkinga, Erik Meijer and Ross Paterson [16, 39], Johan Jeuring and Ralf Hinze [32, 29, 30], and John Hughes [31]. For their inspiration, I am indebted. For further details on the datatype-generic style presented here, see [19, 20] and the above references.

Barry Jay has an alternative approach to datatype-generic programming, which he calls *shape polymorphism* [34, 33]. He and Jens Palsberg have also done some work on a generic representation of the VISITOR pattern [44], but this relies heavily on reflection rather than his work on shape.

For other recent discussions of the meeting between functional and object-oriented views of genericity, see [12, 18].

9. CONCLUSIONS

Design patterns are traditionally expressed informally, using prose, pictures and prototypes. In this paper we have argued that, given the right language features, certain patterns at least could be expressed more usefully as reusable library code. The language features required are *higher-order functions* and *datatype genericity*; for some aspects, *lazy evaluation* also turns out to be helpful. These features are familiar in the world of functional programming; we hope to see them soon in more mainstream programming languages.

10. ACKNOWLEDGEMENTS

This paper elaborates on arguments developed in a course presented while on sabbatical at the University of Canterbury in New Zealand in early 2005, and explored further at tutorials at ECOOP [21] and OOPSLA [22] later that year; the contribution of participants at those venues and at less formal presentations of the same ideas is gratefully acknowledged. The work was carried out as part of the EPSRC-funded *Datatype-Generic Programming* project at Oxford and Nottingham; we thank members of the project for advice and encouragement.

11. REFERENCES

[1] M. Aigner and G. M. Ziegler. *Proofs from The Book*. Springer-Verlag, third edition, 2004.

[2] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[3] R. Backhouse and T. Sheard, editors. *Proceedings of the Workshop on Generic Programming*. Unpublished,

Marstrand, Sweden, June 1998. `http://www.cs.uu.nl/people/johanj/wgp98.html`.

[4] R. C. Backhouse and B. A. Carré. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and Applications*, 15:161–186, 1975.

[5] R. C. Backhouse, P. Jansson, J. Jeuring, and L. G. L. T. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115, 1998.

[6] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.

[7] R. Bird, O. de Moor, and P. Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[8] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Comput. Surv.*, 17(4):471–522, Dec. 1985.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[11] J. C. Dehnert and A. Stepanov. Fundamentals of generic programming. In M. Jazayeri, R. Loos, and D. Musser, editors, *Report of the Dagstuhl Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, Heidelberg, Germany, 2000.

[12] G. Dos Reis and J. Järvi. What is generic programming? In *Library-Centric Software Design*, 2005. OOPSLA workshop.

[13] G. Dos Reis and B. Stroustrup. Specifying C++ concepts. In *Principles of Programming Languages*, 2006.

[14] B. C. dos Santos Oliveira and J. Gibbons. TypeCase: A design pattern for type-indexed functions. In D. Leijen, editor, *Haskell Workshop*, 2005.

[15] J. Edmonds. Matroids and the Greedy Algorithm. *Mathematical Programming*, 1:125–136, 1971.

[16] M. M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, Jan. 1991.

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

[18] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[19] J. Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002.

[20] J. Gibbons. Origami programming. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003.

[21] J. Gibbons. Design patterns as higher-order datatype-generic programs. `http://2005.ecoop.org/8.html`, June 2005. Tutorial presented at ECOOP.

[22] J. Gibbons. Design patterns as higher-order datatype-generic programs. `http://www.oopsla.org/2005/ShowEvent.do?id=121`, Oct. 2005. Tutorial presented at OOPSLA.

[23] J. Gibbons, R. Backhouse, B. Oliveira, and F. Reig. Datatype-generic programming project. `http://web.comlab.ox.ac.uk/oucl/research/pdt/ap/dgp/`, Oct. 2003.

[24] J. Gibbons, G. Hutton, and T. Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1), Apr. 2001. Proceedings of Coalgebraic Methods in Computer Science.

[25] J. Gibbons and G. Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, Sept. 1998.

[26] J. Gibbons, D. Lester, and R. Bird. Enumerating the rationals. *Journal of Functional Programming*, 2006.

[27] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, 1993.

[28] D. Gregor and S. Schupp. Making the usage of STL safe. In J. Gibbons and J. Jeuring, editors, *Generic Programming*. Kluwer Academic Publishers, 2003. Proceedings of the IFIP TC2 Working Conference on Generic Programming, Schloß Dagstuhl, July 2002. ISBN 1-4020-7374-7.

[29] R. Hinze. Polytypic values possess polykinded types. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2000.

[30] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In R. Backhouse and J. Gibbons, editors, *Summer School on Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag, 2003.

[31] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, Apr. 1989.

[32] P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *Principles of Programming Languages*, pages 470–482, 1997.

[33] B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[34] B. Jay and R. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94: 5th European Symposium on Programming, Edinburgh, U.K., April 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer-Verlag, 1994.

[35] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[36] B. Korte, L. Lovász, and R. Schrader. *Greedoids*. Springer-Verlag, 1991.

[37] B. Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.

[38] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[39] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[40] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, 1997.

[41] MITRE Corp. *Ada Reference Manual*. ISO, 2000. IEC 8652.

[42] D. R. Musser and A. A. Stepanov. *The Ada Generic Library linear list processing packages*. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[43] P. Norvig. Design patterns in dynamic programming. In *Object World*, Boston, MA, May 1996. Tutorial slides at `http://norvig.com/design-patterns/`.

[44] J. Palsberg and C. B. Jay. The essence of the Visitor pattern. In *22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[45] A. Pardo. Fusion of recursive programs with computation effects. *Theoretical Comput. Sci.*, 260:165–207, 2001.

[46] A. Pardo. Combining datatypes and effects. In *Advanced Functional Programming*, 2005.

[47] S. Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[48] F. Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns over Types*. PhD thesis, University of Michigan, 1992.

[49] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library*. Addison-Wesley, 2002.

[50] G. T. Sullivan. Advanced programming language features for executable design patterns: Better patterns through reflection. Artificial Intelligence Laboratory Memo AIM-2002-005, Artificial Intelligence Lab, MIT, Mar. 2002.

[51] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, pages 30–50. Springer-Verlag, 2004.

[52] E. Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994.

[53] T. Uustalu and V. Vene. Primitive (co)recursion and course-of-value (co)iteration. Research Report TRITA-IT R 98:02, Dept of Teleinformatics, Royal Institute of Technology, Stockholm, Jan. 1998.

[54] T. Veldhuizen. *Active Libraries and Universal Languages*. PhD thesis, Computer Science, Indiana University, 2004.

[55] V. Vene and T. Uustalu. Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, 47(3):147–161, 1998. 9th Nordic Workshop on Programming Theory.

[56] D. Vytiniotis, G. Washburn, and S. Weirich. An open and shut typecase. In *International Conference on Functional Programming*, 2004.

[57] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[58] P. Wadler. How to solve the reuse problem? Functional programming. In *Internal Conference on Software Reuse*, pages 371–372. IEEE, 1998. `http://doi.ieeecomputersociety.org/10.1109/ICSR.1998.685772`.

[59] P. L. Wadler. The expression problem. Posting to java-genericity mailing list, 12th Nov 1998.

12. APPENDIX: JAVA PROGRAMS

Section 6.5 provides a nearly complete implementation of the document application in a higher-order datatype-generic style; all that is missing is a definition for the spelling corrector *correct*. In contrast, Section 5.5 presents only the outline of a Java implementation of the same application. For completeness, this appendix presents the Java code. This could be omitted on publication of the paper, in case it is accepted, and made available online instead.

12.1 Component

```
public interface Component{
  void accept (Visitor v);
  Iterator getIterator ();
}
```

12.2 Section

```
import java.util.Vector;
import java.util.Enumeration;

public class Section implements Component{
  protected Vector children;
  protected String title;
  public Section (String title){
    children = new Vector ();
    this.title = title;
  }
  public String getTitle (){
    return title;
  }
  public void addComponent (Component c){
    children.addElement (c);
  }
```

```
    public Enumeration getChildren (){
        return children.elements ();
    }
    public Iterator getIterator (){
        return new SectionIterator (this);
    }
    public void accept (Visitor v){
        v.visitSection (this);
    }
}
```

12.3 Paragraph

```
public class Paragraph implements Component{
    protected String body;
    public Paragraph (String body){
        setBody (body);
    }
    public void setBody (String s){
        body = s;
    }
    public String getBody (){
        return body;
    }
    public Iterator getIterator (){
        return new ParagraphIterator (this);
    }
    public void accept (Visitor v){
        v.visitParagraph (this);
    }
}
```

12.4 Iterator

```
public interface Iterator{
    void iterate (Action a);
}
```

12.5 SectionIterator

```
import java.util.Enumeration;

public class SectionIterator implements Iterator{
    protected Section s;
    public SectionIterator (Section s){
        this.s = s;
    }
    public void iterate (Action a){
        for (Enumeration e = s.getChildren ();
                e.hasMoreElements ();){
            ((Component) (e.nextElement ())).
                getIterator ().iterate (a);
        }
    }
}
```

12.6 ParagraphIterator

```
public class ParagraphIterator implements Iterator{
    protected Paragraph p;
    public ParagraphIterator (Paragraph p){
        this.p = p;
    }
    public void iterate (Action a){
        a.apply (p);
    }
}
```

12.7 Action

```
public interface Action{
    void apply (Paragraph p);
}
```

12.8 SpellCorrector

```
public class SpellCorrector implements Action{
    public void apply (Paragraph p){
        p.setBody (correct (p.getBody ()));
    }
    public String correct (String s){
        return s.toLowerCase ();
    }
}
```

12.9 Visitor

```
public interface Visitor{
    void visitParagraph (Paragraph p);
    void visitSection (Section s);
}
```

12.10 PrintVisitor

```
import java.util.Enumeration;
import java.util.Vector;

public class PrintVisitor implements Visitor{

    protected String indent = "";
    protected Vector lines = new Vector ();

    public String [] getResult (){
        String [] ss = new String [0];
        ss = (String []) lines.toArray (ss);
        return ss;
    }
    public void visitParagraph (Paragraph p){
        lines.addElement (indent + p.getBody ());
    }
    public void visitSection (Section s){
        String currentIndent = indent;
        lines.addElement (indent + s.getTitle ());
        for (Enumeration e = s.getChildren ();
                e.hasMoreElements ();){
            indent = currentIndent + "    ";
            ((Component) e.nextElement ()).accept (this);
        }
        indent = currentIndent;
    }
}
```

12.11 Builder

```
public interface Builder{
  int addParagraph (String body, int parent)
    throws InvalidBuilderId;
  int addSection (String title, int parent)
    throws InvalidBuilderId;
}
```

12.12 InvalidBuilderId

```
public class InvalidBuilderId extends Exception{
  public InvalidBuilderId (String reason){
    super (reason);
  }
}
```

12.13 ComponentBuilder

```
import java.util.AbstractMap;
import java.util.HashMap;
public class ComponentBuilder implements Builder{
  protected int nextId = 0;
  protected AbstractMap comps = new HashMap ();
  public int addParagraph (String body, int pId)
      throws InvalidBuilderId{
    return addComponent (new Paragraph (body), pId);
  }
  public int addSection (String title, int pId)
      throws InvalidBuilderId{
    return addComponent (new Section (title), pId);
  }
  public Component getProduct (){
    return (Component) comps.get (new Integer (0));
  }
  protected int addComponent (Component c, int pId)
      throws InvalidBuilderId{
    if (pId < 0){   // root component
      if (comps.isEmpty ()){
        comps.put (new Integer (nextId), c);
        return nextId++;
      }
      else
        throw new InvalidBuilderId
          ("Duplicate root");
    } else {   // non-root
      Component parent = (Component) comps.
        get (new Integer (pId));
      if (parent ≡ null){
        throw new InvalidBuilderId
          ("Non-existent parent");
      } else {
        if (parent instanceof Paragraph){
          throw new InvalidBuilderId
            ("Adding child to paragraph");
        } else {
          Section s = (Section) parent;
          s.addComponent (c);
          comps.put (new Integer (nextId), c);
          return nextId++;
        }
      }
    }
  }
}
```

12.14 PrintBuilder

This is the only class with a non-obvious implementation. It constructs the printed representation (a *String* [ ]) of a *Component* on the fly. In order to do so, it needs to retain some of the tree structure. This is done by maintaining, for each *Component* stored, the unique identifier of its right-most child (or its own identifier, if it has no children). This is stored in the *last* field of the corresponding *Record* in the vector *records*. This vector itself is stored in the order the lines will be returned, that is, a preorder traversal. When adding a new *Component*, it should be placed after the rightmost descendent of its immediate parent, and this is located by following the path of *last* references. (The code would be cleaner if we were to use Java generics to declare *records* as a *Vector*⟨*Record*⟩ rather than a plain *Vector* of *Object*s, but we wish to emphasize that the datatype-genericity discussed in this paper is a different kind of genericity to that provided in Java 1.5.)

```
import java.util.Vector;
public class PrintBuilder implements Builder{
  protected class Record{
    public int id;
    public int last;
    public String line;
    public String indent;
    public Record (int id, int last,
                   String line, String indent){
      this.id = id;
      this.last = last;
      this.line = line;
      this.indent = indent;
    }
  }
  protected Vector records = new Vector ();
  protected Record recordAt (int i){
    return (Record) records.elementAt (i);
  }
  protected int find (int id, int start){
    while (start < records.size () &&
          recordAt (start).id ! = id)
      start++;
    if (start < records.size ())
      return start;
    else
      return − 1;
  }
  protected int nextId = 0;
  protected SpellCorrector c = new SpellCorrector ();
  public int addParagraph (String body, int pid)
      throws InvalidBuilderId{
    return addComponent (c.correct (body), pid);
  }
  public int addSection (String title, int pid)
      throws InvalidBuilderId{
    return addComponent (title, pid);
  }
  public String [ ] getProduct (){
    String [ ] ss = new String [records.size ()];
    for (int i = 0; i < ss.length; i++)
      ss [i] = recordAt (i).indent + recordAt (i).line;
    return ss;
  }
```

```
  protected int addComponent (String s, int pId)
      throws InvalidBuilderId{
    if (pId < 0){    // root component
      if (records.isEmpty ()){
        records.addElement (new Record
          (nextId, nextId, s, ""));
        return nextId++;
      }
      else
        throw new InvalidBuilderId
          ("Duplicate root");
    } else {    // non-root
      int x = find (pId, 0);
      Record r = recordAt (x);
      String indent = r.indent;
      if (x == -1){
        throw new InvalidBuilderId
          ("Non-existent parent");
      } else {
        int y = x;    // ids [x] = ids [y] = pid
        while (r.id ! = r.last){
          y = x;
          x = find (r.last, x);
          r = recordAt (x);
        }    // lasts [y] = lasts [x] = ids [x]
        records.insertElementAt (new Record
          (nextId, nextId, s, indent + "    "), x + 1);
        recordAt (y).last = nextId;
          // lasts [y] = lasts [x + 1] = nextId
        return nextId++;
      }
    }
  }
}
```

12.15 Main

```
public abstract class Main{
  public static void build (Builder b){
    try{
      int rootId = b.addSection ("Doc", -1);
      int sectId = b.addSection ("Sec 1", rootId);
      int subsId = b.addSection ("Subsec 1.1", sectId);
      int id = b.addParagraph ("Para 1.1.1", subsId);
      id = b.addParagraph ("Para 1.1.2", subsId);
      subsId = b.addSection ("Subsec 1.2", sectId);
      id = b.addParagraph ("Para 1.2.1", subsId);
      id = b.addParagraph ("Para 1.2.2", subsId);
      sectId = b.addSection ("Sec 2", rootId);
      subsId = b.addSection ("Subsec 2.1", sectId);
      id = b.addParagraph ("Para 2.1.1", subsId);
      id = b.addParagraph ("Para 2.1.2", subsId);
      subsId = b.addSection ("Subsec 2.2", sectId);
      id = b.addParagraph ("Para 2.2.1", subsId);
      id = b.addParagraph ("Para 2.2.2", subsId);
    }catch (InvalidBuilderId e){
      System.out.println ("Exception: " + e);
    }
  }
  public static void main (String [] args){
    String [] lines;
    if (false){    // build then compute
      ComponentBuilder b = new ComponentBuilder ();
      build (b);
      Component root = b.getProduct ();
      root.getIterator ().iterate (new SpellCorrector ());
      PrintVisitor pv = new PrintVisitor ();
      root.accept (pv);
      lines = pv.getResult ();
    } else {    // computing builder
      PrintBuilder b = new PrintBuilder ();
      build (b);
      lines = b.getProduct ();
    }
    for (int i = 0; i < lines.length; i++)
      System.out.println (lines [i]);
  }
}
```