Datatype-Generic Programming

Jeremy Gibbons

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

Abstract. *Generic programming* aims to increase the flexibility of programming languages, by expanding the possibilities for parametrization — ideally, without also expanding the possibilities for uncaught errors. The term means different things to different people: *parametric polymorphism*, *data abstraction*, *meta-programming*, and so on. We use it to mean polytypism, that is, parametrization by the *shape* of data structures rather than their contents. To avoid confusion with other uses, we have coined the qualified term *datatype-generic programming* for this purpose. In these lecture notes, we expand on the definition of datatype-generic programming, and present some examples of datatype-generic programs. We also explore the connection with *design patterns* in object-oriented programming; in particular, we argue that certain design patterns are just higher-order datatype-generic programs.

1 Introduction

Generic programming is about making programming languages more flexible without compromising safety. Both sides of this equation are important, and becoming more so as we seek to do more and more with computer systems, while becoming ever more dependent on their reliability.

The term 'generic programming' means different things to different people, because they have different ideas about how to achieve the common goal of combining flexibility and safety. To some people, it means *parametric polymorphism*; to others, it means libraries of *algorithms and data structures*; to another group, it means *reflection and meta-programming*; to us, it means *polytypism*, that is, type-safe parametrization by a datatype. Rather than trying to impose our meaning on the other users of the term, or risk confusion by ignoring the other uses, we have chosen to coin the more specific term *datatype-generic programming.* We look in more detail at what we mean by 'datatype-generic programming', and how it relates to what others mean by 'generic programming', in Section 2.

Among the various approaches to datatype-generic programming, one is what we have called elsewhere *origami programming* [38], and what others have variously called *constructive algorithmics* [12, 123], *Squiggol* [93], *bananas and lenses* [101], the *Bird-Meertens Formalism* [122, 52], and the *algebra of programming* [9], among other names. This is a style of functional (or relational) programming based on maps, folds, unfolds and other such higher-order structured recursion operators. Malcolm [92], building on earlier theoretical work by Hagino [56], showed how the existing ad-hoc datatype-specific recursion operators (maps and folds on lists, on binary trees, and so on) could be unified datatype-generically. We explain this school of programming in Section 3.

The origami approach to datatype-generic programming offers a number of benefits, not least of which is the support it provides for reasoning about recursive programs. But one of the reasons for our interest in the approach is that it seems to offer a good way of capturing precisely the essence of a number of the so-called Gang of Four *design patterns*, or reusable abstractions in object-oriented software [35]. This is appealing, because without some kind of datatype-generic constructs, these patterns can only be expressed extra-linguistically, 'as prose, pictures, and prototypes', rather than captured in a library, analysed and reused. We argue this case in Section 4, by presenting higher-order datatype-generic programs capturing ORIGAMI, a small suite of patterns for recursive data structures.

A declarative style of origami programming seems to capture well the computational structure of at least some of these patterns. But because they are usually applied in an imperative setting, they generally involve impure aspects too; a declarative approach does not capture those aspects well. The standard approach the functional programming community now takes to incorporating impure features in a pure setting is by way of *monads* [105, 135], which elegantly model all sorts of impure effects such as state, I/O, exceptions and non-determinism. More recently, McBride and Paterson have introduced the notion of *idiom* or *applicative functor* [95], a slight generalization of monads with better compositional properties. One consequence of their definitions is a datatype-generic means of *traversing* collections 'idiomatically', incorporating both pure accumulations and impure effects. In Section 5, we explore the extent to which this construction offers a more faithful higher-order datatype-generic model of the ITERATOR design pattern specifically.

These lecture notes synthesize ideas and results from earlier publications, rather than presenting much that is new. In particular, Section 3 is a summary of two earlier sets of lectures [37, 38]; Section 4 recaps the content of a tutorial presented at ECOOP [39] and OOPSLA [40], and subsequently published in a short paper [41]; Section 5 reports on some more recent joint work with Bruno Oliveira [44]. Much of this work took place within the EPSRC-funded *Datatype-Generic Programming* project at Oxford and Nottingham, of which this Spring School marks the final milestone.

2 Generic programming

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 one started with. Ideally, the abstraction increases expressivity, when some instantiations of the parameter yield new programs in addition to the original ones; otherwise, all one has gained is some elimination of duplication and a warm fuzzy feeling. 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. (This is analogous to the so-called *AI effect*: Rodney Brooks, director of MIT's Artificial Intelligence Laboratory, quoted in [79], observes that 'Every time we figure out a piece of [AI], it stops being magical; we say, "Oh, that's just a computation" '.) 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-reference parameter-passing mechanisms for arguments to procedures, as found in Pascal [74], look like 'generic programming' to an assembly-language programmer with no such tools at their disposal.

In this section, we review a number of interpretations of 'genericity' in terms of the kind of parametrization they support. Parametrization by *value* is the kind of parameter-passing mechanism familiar from most programming languages, and while (as argued above) this would not normally be considered 'generic programming', we include it for completeness; parametrization by *type* is what is normally known as polymorphism; parametrization by *function* is sometimes called 'higher-order programming', and is really just parametrization by value where the values are functions; parametrization by *structure* involves passing 'modules' with a varying private implementation of a fixed public signature or interface; parametrization by *property* is a refinement of parametrization by structure, whereby operations of the signature are required to satisfy some laws; parametrization by *stage* allows programs to be partitioned, with meta-programs that generate object programs; and parametrization by *shape* is to parametrization by type as 'by function' is to 'by value'.

2.1 Genericity by value

One of the first and most fundamental techniques that any programmer learns is how to parametrize computations by values. Those old enough to have been brought up on structured programming are likely to have been given exercises to write programs to draw simple ASCII art: Whatever the scenario, students soon realise the futility of hard-wiring fixed behaviour into programs:

```
procedure Triangle4;
begin
  WriteString ("*"); WriteLn;
  WriteString ("**"); WriteLn;
  WriteString ("***"); WriteLn;
  WriteString ("****"); WriteLn
end;
```

and the benefits of abstracting that behaviour into parameters:

```
procedure Triangle (Side : cardinal);
begin
  var Row, Col : cardinal;
  for Row := 1 to Side do begin
    for Col := 1 to Row do WriteChar ('*');
    WriteLn
  end
end
```

Instead of a parameterless program that always performs the same computation, one ends up with a program with formal parameters, performing different but related computations depending on the actual parameters passed: a *function*.

2.2 Genericity by type

Suppose that one wants a datatype of lists of integers, and a function to append two such lists. These are written in Haskell [112] as follows:

```
data ListI = NilI | ConsI Integer ListI
appendI :: ListI → ListI → ListI
appendI NilI          ys = ys
appendI (ConsI x xs) ys = ConsI x (appendI xs ys)
```

Suppose in addition that one wanted a datatype and an append function for lists of characters:

```
data ListC = NilC | ConsC Char ListC
appendC :: ListC → ListC → ListC
appendC NilC          ys = ys
appendC (ConsC x xs) ys = ConsC x (appendC xs ys)
```

It is tedious to repeat similar definitions in this way, and it doesn't take much vision to realise that the repetition is unnecessary: the definitions of the datatypes *ListI* and *ListC* are essentially identical, as are the definitions of the functions *appendI* and *appendC*. Apart from the necessity in Haskell to choose distinct names, the only difference in the two datatype definitions is the type of list elements, *Integer* or *Char*. Abstracting from this hard-wired constant leads to a single *polymorphic* datatype parametrized by another type, the type of list elements:

```
data List a = Nil | Cons a (List a)
```

(The term 'parametric datatype' would probably be more precise, but 'polymorphic datatype' is well established.) Unifying the two list datatypes in this way unifies the two programs too, into a single polymorphic program:

```
append :: List a → List a → List a
append Nil          ys = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

There is a precise technical sense in which the process of abstraction by which one extracts type parameters reflects that by which one extracts value parameters. In Haskell, our definition of the polymorphic datatype *List* introduces a polymorphic value *Nil* of type *List a* for any type *a*; in the polymorphic lambda calculus [51, 116], the polymorphism is made manifest in a type parameter: *Nil* would have type $\Lambda a.\,List\ a$ abstracted on the list element type a, and $Nil\ \tau$ would have type $List\ \tau$ for some specific element type τ.

This kind of type abstraction is called *parametric polymorphism.* It entails that the instantiated behaviour is *uniform* in the type parameter, and cannot depend on what actual parameter it is instantiated to. Informally, this means that a polymorphic function cannot examine elements of polymorphic types, but can merely rearrange them. Formally, this intuition was captured by Reynolds in his *abstraction theorem* [117], generalized by Wadler to the *parametricity theorem* and popularized under the slogan 'Theorems for Free' [132]. In the case of *append*, (a corollary of) the free theorem states that, for any function *a* of the same type as *append*,

$$a\ (mapL\ f\ xs)\ (mapL\ f\ ys) = mapL\ f\ (a\ xs\ ys)$$

where the function *mapL* (explained in Section 2.3 below) applies its first argument, a function, to every element of its second argument, a list.

Related to but not quite the same as parametric polymorphism is what Cardelli and Wegner [16] call *inclusion polymorphism.* This is the kind of polymorphism arising in object-oriented languages. Consider, for example, the following Java method:

```
public void addObserver (Observer obs){
  observers.addElement (obs);
}
```

This method takes a parameter *obs* of varying type, as does the Haskell function *append*; moreover, it behaves uniformly in the type of the actual parameter passed. However, it doesn't accept parameters of an arbitrary type, like *append* does, but only parameters whose type is included in the type *Observer*. (Alternatively, one could say that the method takes a parameter exactly of type *Observer*, but that subtypes of *Observer* are subsumed within this type.) We discuss the relationship between inclusion polymorphism and parametric polymorphism, and between these two and so-called ad-hoc forms of polymorphism, in Section 2.8 below.

One well-established interpretation of the term 'generic programming' is exactly as embodied by parametric polymorphism and inclusion polymorphism. Cardelli and Wegner [16, p. 475] state that 'the functions that exhibit parametric polymorphism are [...] called generic functions', and give $length :: List\ a \rightarrow Integer$ as an example. Paradigmatic languages exhibiting parametric polymorphism are ML [104] and Haskell [112], which provide (variations on) Hindley–Milner–Damas typing [103, 23]. These have influenced the 'generics' in recent versions of Java [13] and C# [80]. (On the other hand, CLOS [81] also uses the term 'generic function', but in a sense related to inclusion polymorphism.)

Ada 83 [1] had a notion of generics, by which procedures and 'packages' (modules) can be parametrized by values, types, procedures (which gives a kind of higher-order parametrization, as discussed in Section 2.3) and packages (which gives what we call parametrization by structure, and discuss in Section 2.4). For example, the code below shows: (1) the declaration of a generic subprogram *Swap*, parametrized by a type *Element*; (2) the generic body of the subprogram, which makes use of the formal parameter *Element*; and (3) the instantiation of the generic unit to make a non-generic subprogram that may be used in the same way as any other subprogram.

```
generic
  type Element is private;
procedure Swap (X, Y : in out Element);              -- (1)
procedure Swap (X, Y : in out Element) is             -- (2)
  Z : constant Element := X;
begin
  X := Y;
  Y := Z;
end Swap;
procedure SwapInteger is new Swap (Integer);    -- (3)
```

However, Ada generic units are templates for their non-generic counterparts, as are the C++ templates they inspired, and cannot be used until they are instantiated; Cardelli and Wegner observe that this gives the advantage that instantiation-specific compiler optimizations may be performed, but aver that 'in true polymorphic systems, code is generated only once for every generic procedure' [16, p. 479].

2.3 Genericity by function

Higher-order programs are programs parametrized by other programs. We mentioned above that Ada 83 generics allow parametrization by procedure; so do languages in the Algol family [111, 108, 138, 128]. However, the usefulness of higher-order parametrization is greatly reduced in these languages by the inability to express actual procedure parameters anonymously in place. Higher-order parametrization comes into its own in functional programming, which promotes exactly this feature: naturally enough, making functions first-class citizens.

Suppose one had strings, represented as lists of characters, that one wanted to convert to uppercase, perhaps for the purpose of normalization:

```
stringToUpper :: List Char → List Char
stringToUpper Nil          = Nil
stringToUpper (Cons x xs) = Cons (toUpper x) (stringToUpper xs)
```

where *toUpper* converts characters to uppercase. Suppose also that one had a list of integers for people's ages, which one wanted to classify into young and old, represented as booleans:

$$\begin{aligned}&\mathit{classifyAges} :: \mathit{List\ Integer} \to \mathit{List\ Bool}\\&\mathit{classifyAges\ Nil} = \mathit{Nil}\\&\mathit{classifyAges}\ (\mathit{Cons\ x\ xs}) = \mathit{Cons}\ (x < 30)\ (\mathit{classifyAges\ xs})\end{aligned}$$

These two functions, and many others, follow a common pattern. What differs is in fact a value, but one that is higher-order rather than first-order: the function to apply to each list element, which in the first case is the function *toUpper*, and in the second is the predicate ($<$30). What is common between the two is the function *mapL*, mentioned in Section 2.2 above:

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

We treat this kind of parametrization separately from parametrization by first-order value, because it has far-reaching consequences. Among other things, it lets programmers express control structures within the language, rather than having to extend the language. For example, one might already consider *mapL* to be a programmer-defined control construct. For another example, recall the parametrically polymorphic *append* function from Section 2.2:

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

A second function, *concat*, concatenates a list of lists into one long list:

$$\begin{aligned}&\mathit{concat} :: \mathit{List}\ (\mathit{List}\ a) \to \mathit{List}\ a\\&\mathit{concat\ Nil} = \mathit{Nil}\\&\mathit{concat}\ (\mathit{Cons\ xs\ xss}) = \mathit{append\ xs}\ (\mathit{concat\ xss})\end{aligned}$$

A third sums a list of integers:

$$\begin{aligned}&\mathit{sum} :: \mathit{List\ Integer} \to \mathit{Integer}\\&\mathit{sum\ Nil} = 0\\&\mathit{sum}\ (\mathit{Cons\ x\ xs}) = x + \mathit{sum\ xs}\end{aligned}$$

Each of the three programs above traverses its list argument in exactly the same way. Abstracting from their differences allows us to capture that control structure as a pattern of recursion. The common pattern is called a 'fold':

$$\begin{aligned}&\mathit{foldL} :: b \to (a \to b \to b) \to \mathit{List}\ a \to b\\&\mathit{foldL}\ n\ c\ \mathit{Nil} = n\\&\mathit{foldL}\ n\ c\ (\mathit{Cons\ x\ xs}) = c\ x\ (\mathit{foldL}\ n\ c\ \mathit{xs})\end{aligned}$$

(We write the suffix 'L' to denote an operation over lists; Sections 2.7 and 3 discuss generalizations to other datatypes. It so happens that this function is equivalent to the function *foldr* — 'fold from the right' — in the Haskell standard library [112].) Instances of *foldL* replace the list constructors *Nil* and *Cons* with supplied arguments:

$$\begin{aligned} &\mathit{append}\ \mathit{xs}\ \mathit{ys} &&= \mathit{foldL}\ \mathit{ys}\ \mathit{Cons}\ \mathit{xs} \\ &\mathit{concat} &&= \mathit{foldL}\ \mathit{Nil}\ \mathit{append} \\ &\mathit{sum} &&= \mathit{foldL}\ 0\ (+) \end{aligned}$$

In fact, *mapL* turns out to be another instance of *foldL*:

$$\mathit{mapL}\ f = \mathit{foldL}\ \mathit{Nil}\ (\mathit{Cons} \circ f)$$

where $\circ$ is function composition (itself another higher-order operator).

2.4 Genericity by structure

Perhaps the most popular interpretation of the term 'generic programming' is as embodied in the C++ Standard Template Library, an object-oriented class library providing *containers*, *iterators* and *algorithms* for many datatypes [4]. Indeed, some writers have taken the STL style as the definition of generic programming; for example, Siek et al. [120] define generic programming as 'a methodology for program design and implementation that separates data structures and algorithms through the use of abstract requirement specifications'.

As the name suggests, the STL 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 [107].) Within the STL community more than any other, it is considered essential that genericity imposes no performance penalty [25, 129].

The *containers* that are provided 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 be incremented, copied, assigned, compared for equality, and read from — that is, used as r-values), *output iterators* (which are similar, but can only be written to — that is, used as l-values), *forwards iterators* (which refine both input and output iterators), *bidirectional 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. The algorithms provided in the STL include many general-purpose operations such as searching, sorting, filtering, and so on. Rather than operating directly on containers, an algorithm takes one or more iterators as parameters; the algorithm is *generic*, in the sense that it applies to any container that supports the appropriate kind of iterator.

For example, here is a code fragment (taken from [4, §1.1]) implementing a simplified version of the Unix `sort` utility.

```
int main () {
  vector⟨string⟩ v;
  string tmp;
  while (getline (cin, tmp))
    v.push_back (tmp);
  sort (v.begin (), v.end ());
  copy (v.begin (), v.end (), ostream_iterator⟨string⟩(cout, "\n"));
}
```

It shows a container v (a vector of strings); applications of generic algorithms (*sort* and *copy*); and a pair of iterators ('pointers' *v.begin* () and *v.end* () to the beginning and just past-the-end of the vector v) mediating between them.

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) is called a *concept*. A concept encapsulates the operations required of a formal type parameter and provided by an actual type parameter. Algorithms and containers are parametrized by the concept, and instantiated by passing a *structure* that implements the concept. For example, the STL's 'input iterator' concept encompasses pointer-like types which support comparison for equality, copying, assignment, reading, and incrementing. The success of the STL lies pretty much in the careful choice of such concepts as an organizing principle for a large library; as Siek and Lumsdaine [121] explain, the same principle has worked for many other C++ class libraries too.

The C++ template mechanism provides no means to define a concept explicitly; it is merely an informal artifact rather than a formal construct. (However, work is proceeding to formalize concepts as language constructs; see for example [54, 121, 53].) In that sense, it is a retrograde step from earlier languages supporting data abstraction. For instance, Liskov's CLU language [89] from the mid-1970s had a **where** clause, for specifying the requirements (names and signatures) on a type parameter to a *cluster*; the following declaration [89, p13] for a cluster *set* parametrized by a type t states that t must support a binary predicate *equal*.

```
set = cluster [t : type] is create, member, size, insert, delete, elements
  where t has equal : proctype (t, t) returns (bool)
```

This retrograde step is somewhat ironic, since CLU's clusters were, via Ada generics, the inspiration for the C++ template mechanism in the first place.

The notion in Haskell analogous to C++'s concepts, and the basis for current proposals to to provide linguistic support for concepts in C++ [53], is the *type class*, which also captures the requirements required of and provided by types, but which is formally part of the language. For example, a function *sort* in Haskell might have the following type:

$$sort :: Ord\ a \Rightarrow List\ a \rightarrow List\ a$$

The constraint '$Ord\ a \Rightarrow$' is a *type class context*; the function *sort* is not parametrically polymorphic, because it is not applicable to all types of list element, only those in the type class *Ord*. The type class *Ord* includes exactly those types that support the operation $\leqslant$, and might be defined in Haskell as follows:

```
class Ord a where
  (≤) :: a → a → Bool
```

(The actual definition is more complex than this [112]; but this simpler version serves for illustration.) Various types are instance of the type class, by virtue of supporting a comparison operation:

```
instance Ord Integer where
  (m ≤ n) = isNonNegative (n − m)
```

Attempting to apply $\leqslant$ to two values of some type that is not in the type class *Ord*, or *sort* to a list of such values, is a type error, and is caught statically. In contrast, while the equivalent error using the C++ STL 'less-than comparable' concept is still a statically caught type error, it is caught at template instantiation time, since there is no way of declaring the uninstantiated template's dependence on the concept.

As we stated above, the kind of polymorphism provided by C++ STL concepts and Haskell type classes is not parametric, because it is not universal. For the same reason, neither is it inclusion polymorphism, even though C++ concepts and Haskell type classes both form hierarchies. In fact, the member functions of the Haskell type class, such as the operation $(\leqslant) :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Bool$, are *ad-hoc polymorphic*, which is to say non-uniform: there is no requirement, and indeed it is generally not the case, that definitions of $\leqslant$ on different types will be implemented using the same code.

2.5 Genericity by property

We have seen that generic programming in the C++ sense revolves around concepts, which are abstractions of the requirements on and provisions of a type parameter, specifically in terms of the operations available. In fact, in typical usage, concepts are more elaborate than this; as well as signatures of operations, the concept might specify the laws these operations satisfy, and non-functional characteristics such as the asymptotic complexities of the operations in terms of time and space. For example, the 'less-than comparable' concept in the STL stipulates that the ordering should be a partial ordering, and the 'random-access iterator' concept stipulates that addition to and subtraction from the pointer should take constant time. Correctness of operation signatures can be *verified* at instantiation time, and this information is useful to a compiler, for example in providing efficient dispatching. The laws satisfied by operations and non-functional characteristics such as complexity cannot be verified in general, although testing frameworks such as QuickCheck [18] and JUnit [34] can go some way towards *validation*; this information nevertheless might still be useful to a sophisticated optimizing compiler.

In the Haskell setting, the formal part of a type class declaration states the names and signatures of the operations provided, and the equally important but informal accompanying documentation may stipulate additional properties, typically in the form of axioms. For example, the *Ord* type class might stipulate that $\leqslant$ forms a total order or a total preorder; we make use later of a type class *Monoid*:

```
class Monoid m where
  ∅   :: m
  (⊕) :: m → m → m
```

with the usual monoid laws:

$$\begin{aligned} x \oplus (y \oplus z) &= (x \oplus y) \oplus z \\ x \oplus \emptyset &= x \\ \emptyset \oplus x &= x \end{aligned}$$

In Section 3, we make use of a two-parameter version of the following one-parameter *Functor* type class:

```
class Functor f where
  fmap :: (a → b) → (f a → f b)
```

(Strictly speaking, *Functor* is a *constructor class* rather than a type class, since its members are type constructors rather than base types: in Haskell terminology, 'types of kind $* \rightarrow *$' rather than 'types of kind $*$', where kinds classify types in the same way that types classify values. But in these lecture notes, we use the term 'type class' even for classes of type constructors.) The intention is that this class contains types supporting functions like *mapL*:

```
instance Functor List where
  fmap = mapL
```

The informal intention 'functions like *mapL*' can be captured more formally in terms of the functor laws:

$$\begin{aligned} \mathit{fmap}\ (f \circ g) &= \mathit{fmap}\ f \circ \mathit{fmap}\ g \\ \mathit{fmap}\ \mathit{id} &= \mathit{id} \end{aligned}$$

In Section 5, we generalize the *Monad* type class, a subclass of *Functor*:

```
class Functor m ⇒ Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
```

(The Haskell 98 standard library [112] omits the *Functor* context, but without any increase in generality, since the operation *fmap* can be reconstructed from *return* and ≫=.) Instances of *Monad* correspond to types of 'computations with impure effects'. The exception monad is a simple instance; a 'computation yielding an *a*' might fail.

```
data Maybe a = Nothing | Just a

foldM :: b → (a → b) → Maybe a → b
foldM y f Nothing = y
foldM y f (Just x) = f x

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)

instance Monad Maybe where
  return a  = Just a
  mx ≫= k = foldM Nothing k mx

raise :: Maybe a                          -- raise an exception
raise = Nothing

trycatch :: (a → b) → b → Maybe a → b     -- handle an exception
trycatch f y = foldM y f
```

Another instance is the state monad, in which a 'computation yielding an *a*' also affects a state of type *s*, amounting to a function of type $s \to (a, s)$:

```
newtype State s a = St (s → (a, s))

runSt :: State s a → s → (a, s)
runSt (St f) = f

instance Functor (State s) where
  fmap f mx = St (λs → let (a, s′) = runSt mx s in (f a, s′))

instance Monad (State s) where
  return a  = St (λs → (a, s))
  mx ≫= k = St (λs → let (a, s′) = runSt mx s in runSt (k a) s′)

put :: s → State s ()   -- write to the state
put s = St (λ_ → ((), s))

get :: State s s        -- read from the state
get = St (λs → (s, s))
```

Haskell provides a convenient piece of syntactic sugar called '**do** notation' [134], allowing an imperative style of programming for monadic computations. This is defined by translation into expressions involving *return* and ≫=; a simplified version of the full translation [112, §3.14] is as follows:

```
do {mx}              = mx
do {x ← mx; stmts}   = mx ≫= λx  → do {stmts}
do {     mx; stmts}  = mx ≫= λ() → do {stmts}
```

In addition to the laws inherited from the *Functor* type class, a *Monad* instance must satisfy the following three laws:

$$\begin{aligned}
&\mathit{return}\ a \gg\!= k && = k\ a \\
&m \gg\!= \mathit{return} && = m \\
&m \gg\!= (\lambda x \rightarrow k\ x \gg\!= h) && = (m \gg\!= k) \gg\!= h
\end{aligned}$$

The first two are kinds of unit law, and the last one a kind of associative law. Their importance is in justifying the use of the imperative style of syntax; for example, they justify flattening of nested blocks:

$$\mathbf{do}\,\{\,p; \mathbf{do}\,\{\,q; r\,\}; s\,\} = \mathbf{do}\,\{\,p; q; r; s\,\}$$

We leave it to the reader to verify that the *Maybe* and *State* instances of the *Monad* class do indeed satisfy these laws.

More elaborate examples of genericity by property arise from more sophisticated mathematical structures. For example, Horner's rule for polynomial evaluation [21] can be parametrized by a semiring, an extremal path finder by a regular algebra [5, 7], and a greedy algorithm by a matroid or greedoid structure [27, 85]. Mathematical structures are fertile grounds for finding more such examples; the Axiom programming language for computer algebra [73, 14] now has a library of over 10,000 'domains' (types).

Whereas genericity by structure is an outcome of the work on abstract datatypes [90], genericity by property follows from the enrichment of that work to include equational constraints, leading to *algebraic specifications* [28, 29], as realised in languages such as Larch [55] and Casl [20, 8].

2.6 Genericity by stage

Another interpretation of the term 'generic programming' covers various flavours of *metaprogramming*, that is, the development of programs that construct or manipulate other programs. This field encompasses *program generators* such as `lex` [75, §A.2] and `yacc` [75, §A.3], *reflection techniques* allowing a program (typically in a dynamically typed language) to observe and possibly modify its structure and behaviour [33], *generative programming* for the automated customization, configuration and assembly of components [22], and *multi-stage programming* for partitioning computations into phases [125].

For example, an *active library* [129] might perform domain-specific optimizations such as unrolling inner loops: rather than implementing numerous slightly different components for different loop bounds, the library could provide a single *generic metaprogram* that specializes to them all. A compiler could even be considered a generative metaprogram: rather than writing machine code directly, the programmer writes meta-machine code in a high-level language, and leaves the generation of the machine code itself to the compiler.

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; as noted above, several high-performance numerical libraries rely on templates' generative properties [129]. The template instantiation mechanism turns out to be Turing complete; Unruh [126] gives the unsettling example of a program whose compilation yields the prime numbers as error messages, Czarnecki and Eisenecker [22] show the Turing-completeness of the template mechanism by implementing a rudimentary Lisp interpreter as a template meta-program, and Alexandrescu [3] presents a *tour de force* of unexpected applications of templates.

2.7 Genericity by shape

Recall the polymorphic datatype *List* introduced in Section 2.2, and the corresponding polymorphic higher-order function *foldL* in Section 2.3; recall also the polymorphic datatype *Maybe* and higher-order function *foldM* from Section 2.5. One might also have a polymorphic datatype of binary trees:

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

The familiar process of abstraction from a collection of similar programs would lead one to identify the natural pattern of recursion 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:

```
reflect :: Btree a → Btree a
reflect = 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
```

We have seen that each kind of parametrization allows some recurring patterns to be captured. For example, parametric polymorphism unifies commonality of computation, abstracting from variability in irrelevant typing information, and 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, nor module signatures 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 3 below, we explain the definition of a datatype-generic operation *fold* with the following type:

$$\mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to \mathit{Fix}\ s\ a \to b$$

Here, in addition to the type a of collection elements and the fold body (a function of type $s\ a\ b \to b$), 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 $\mathit{Fix}\ s\ a$ is isomorphic to $\mathit{List}\ a$, and for another instantiation it is isomorphic to $\mathit{Tree}\ a$. (So the parametrization is strictly speaking not by the recursive datatype *List* itself, but by the bifunctor s that yields the shape of *List*s.) The same shape parameter also determines the type of the fold body, supplied as an argument with which to replace the constructors.

The *Datatype-Generic Programming* project at Oxford and Nottingham [43] has been investigating programs parametrized by datatypes, that is, by type constructors such as ‘list of’ and ‘tree of’. Such programs might be *parametrically datatype-generic*, as with *fold* above, when the behaviour is uniform in the shape parameter. Since the shape parameter is of higher kind, this is a higher-order parametricity property, but it is of the same flavour as first-order parametricity [117, 132], stating a form of coherence between instances of *fold* for different shapes. A similar class of programs is captured by Jay’s theory of *shapely polymorphism* [72].

Alternatively, such programs might be *ad-hoc datatype-generic*, when the behaviour exploits that shape in some essential manner. Typical examples of the latter are pretty printers and marshallers; these can be defined once and for all for lists, trees, and so on, in a typesafe way, but not in a way that guarantees any kind of uniformity in behaviour at the instances for different shapes. This approach to datatype genericity has been variously called *polytypism* [68], *structural polymorphism* [118] or *typecase* [131, 26], and is the meaning given to ‘generic programming’ by the Generic Haskell [60, 91] team. Whatever the name, functions are defined inductively by case analysis on the structure of datatypes; the different approaches differ slightly in the class of datatypes covered. For example, here is a Generic Haskell definition of datatype-generic encoding to a list of bits.

```
type Encode{[*]}      t = t → [Bool]
type Encode{[k → l]} t = ∀a.  Encode{[k]} a → Encode{[l]} (t a)
encode{|t :: k|}                 :: Encode{[k]} t
encode{|Char|} c                 = encodeChar c
encode{|Int|} n                  = encodeInt n
encode{|Unit|} unit              = [ ]
encode{|:+:|} ena enb (Inl a)    = False : ena a
encode{|:+:|} ena enb (Inr b)    = True : enb b
encode{|:×:|} ena enb (a :×: b)  = ena a ++ enb b
```

The generic function *encode* works for any type constructed from characters and integers using sums and products; these cases are defined explicitly, and the cases for type abstraction, application and recursion are generated automatically. Note that instances of *encode* are very different for different type parameters, not even taking the same number of arguments. In fact, the instances have different *kinds* (as mentioned in Section 2.5, and discussed further in Hinze and others' two chapters [61, §3.1] and [63, §2.1] in this volume), and *type-indexed values have kind-indexed types* [57].

As we have seen, ad-hoc datatype-generic definitions are typically given by case analysis over the structure of types. One has the flexibility to define different behaviour in different branches, and maybe even to customize the behaviour for specific types; consequently, there is no guarantee or check that the behaviours in different branches conform, except by type. This is in contrast to the parametrically datatype-generic definition of *fold* cited above; there, 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 these lecture notes, we concentrate on parametric datatype-generic definitions where possible. In fact, it is usually the case that one must provide an ad-hoc datatype-generic hook initially, but then one can derive a number of parametrically datatype-generic definitions from this. In Sections 3 and 4, we suppose an (ad-hoc) datatype-generic operator *bimap*, and from this derive various (parametrically) datatype-generic recursion operators. In Section 5.2 we suppose a different (ad-hoc) datatype-generic operator *traverse*, and from this derive various (parametrically) datatype-generic traversal operators. Clarke and Löh [19] use the name *generic abstractions* for parametrically datatype-generic functions defined in terms of ad-hoc datatype-generic functions.

Datatype genericity is different from various other 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.

2.8 Universal vs ad-hoc genericity

Strachey's seminal notes on programming languages [124] make the fundamental distinction between *parametric* polymorphism, in which a function works uniformly on a range of types, usually with a common structure, and *ad-hoc* polymorphism, in which a function works (or appears to work) on several different types, but these need not have a common structure, and the behaviour might be different at different types.

Cardelli and Wegner [16] refine this distinction. They rename the former to *universal* polymorphism, and divide this into *parametric* polymorphism again and *inclusion* polymorphism. The difference between the two arises from the different ways in which a value may have multiple types: with parametric polymorphism, values and functions implicitly or explicitly take a type parameter, as discussed in Section 2.2; with inclusion polymorphism, types are arranged into a hierarchy, and a value of one type is simultaneously a value of all its supertypes. Cardelli and Wegner also refine ad-hoc polymorphism into *overloading*, a syntactic mechanism in which the same function name has different meanings in different contexts, and *coercion*, in which a function name has just one meaning, but semantic conversions are applied to arguments where necessary.

The uppermost of these distinctions can be applied to other kinds of parameter than types, at least informally. For example, one can distinguish between *universal parametrization* by a number, as in the structured program in Section 2.1 to draw a triangle of a given size, and *ad-hoc parametrization* by a number, as in 'press 1 to listen to the message again, press 2 to return the call, press 3 to delete the message...'-style interfaces. In the former, there is some coherence between the instances for different numbers, but in the latter there is not. For another example, sorting algorithms that use only comparisons obey what is known as the *zero-one principle* [21]: if they work correctly on sequences of numbers drawn from the set $\{0, 1\}$, then they work correctly on arbitrary number sequences (and more generally, where the element type is linearly ordered). Therefore, sorting algorithms defined using only comparisons are universally parametric in the list elements [24], whereas sorting algorithms using other operations (such as radix sort, which depends on the 'digits' or fields of the list elements) are ad-hoc parametric, and proving their correctness requires more effort. *Data independence* techniques in model checking [87, 88] are a third illustration. All of these seem to have some relation to the notion of naturality in category theory, and (perhaps not surprisingly) Reynolds' notion of parametricity. For Cardelli and Wegner, 'universal polymorphism is considered true polymorphism, whereas ad-hoc polymorphism is some kind of apparent polymorphism whose polymorphic character disappears at close range'; by the same token, we might say that *universal parametrization is truly generic, whereas ad-hoc parametrization is only apparently generic.*

2.9 Another dimension of classification

Backhouse et al. [7] suggest a second dimension of classification of parametrization: not only in terms of the varieties of entity that can be abstracted, but also in terms of what support is provided for this abstraction — the varieties of construct from which these entities may be abstracted, and whether instances of those entities can be expressed anonymously in place, or must be defined out of line and referred to by name.

For an example of the second kind of distinction, consider values and types in Haskell 98: values can be parametrized by values (for example, a function *preds* taking an integer n to the list $[n, n-1, ..., 1]$ can be considered as a list parametrized by an integer), types can be parametrized by types (for example, the polymorphic list type $[a]$ is parametrized by the element type a), values can be parametrized by types (for example, the empty list $[\,]$ is polymorphic, and really stands for a value of type $[a]$ for any type a), but types cannot easily be parametrized by values (to capture a type of 'lists of length n', one requires dependent types [96], or some lightweight variant such as generalized algebraic datatypes [113, 119]). We referred in Section 2.3 to an example of the third kind of distinction: although procedures in languages in the Algol family can be parametrized by other procedures, actual procedure parameters must be declared out of line and passed by name, rather than being defined on the fly as lambda expressions.

3 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 [92, 101, 6, 9, 37]. We saw a glimpse of this field in Sections 2.3, 2.5 and 2.7, with the definitions of *foldL*, *foldM* 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 any 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 [48]; 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 all of these. There are many other datatype-generic patterns of computation that we might also have considered: paramorphisms [98], apomorphisms [130], histomorphisms and futumorphisms [127], metamorphisms [42], dynamorphisms [78], destroy [36], and so on.

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. We recently coined the term *origami programming* [38] for this approach to datatype-generic programming, because of its dependence on folds and unfolds.

3.1 Maps and folds on lists

Here is the datatype of lists again.

```
data List a = Nil | Cons a (List a)
```

The ‘map’ operator for a datatype applies a given function to every element of a data structure. In Section 2.3, we saw the (higher-order, polymorphic, but list-specific) map operator for lists:

```
mapL :: (a → b) → (List a → List b)
mapL f Nil          = Nil
mapL f (Cons x xs) = Cons (f x) (mapL f xs)
```

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 from Section 2.3:

```
foldL :: b → (a → b → b) → List a → b
foldL e f Nil          = e
foldL e f (Cons x xs) = f x (foldL e f xs)
```

As a simple application of *foldL*, 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.

```
filterL :: (a → Bool) → List a → List a
filterL p = foldL Nil (add p)
  where add p x xs = if p x then Cons x xs else xs
```

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

3.2 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 becomes clearer with the datatype-generic version we present in Section 3.8.

```
unfoldL ::  (b → Bool) → (b → a) → (b → b) → b → List a
unfoldL p f g x
   = if p x then Nil
            else  Cons (f x) (unfoldL p f g (g x))
```

For example, here are two instances. The function *preds* returns the list of predecessors of an integer (which will be an infinite list if that integer is negative); 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.

```
preds :: Integer → List Integer
preds = unfoldL (0 ==) id pred where pred n = n − 1
```

```
takeWhile :: (a → Bool) → List a → List a
takeWhile p = unfoldL (firstNot p) head tail
  where firstNot p Nil          = True
        firstNot p (Cons x xs) = not (p x)
```

3.3 Origami for binary trees

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

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

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

```
mapT :: (a → b) → (Tree a → Tree b)
mapT f Empty          = Empty
mapT f (Node x xs ys) = Node (f x) (mapT f xs) (mapT f ys)
```

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

```
foldT :: b → (a → b → b → b) → Tree a → b
foldT e n Empty          = e
foldT e n (Node x xs ys) = n x (foldT e n xs) (foldT e n ys)
```

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

```
inorder :: Tree a → List a
inorder = foldT Nil glue

glue x xs ys = append xs (Cons x ys)
```

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

```
unfoldT ::  (b → Bool) → (b → a) → (b → b) → (b → b) → b → Tree a
unfoldT p f g h x
  = if p x then Empty
           else  Node (f x) (unfoldT p f g h (g x))
                            (unfoldT p f g h (h x))
```

For example, the Calkin–Wilf tree, illustrated in Figure 1, contains each of the positive rationals exactly once:

```
cwTree :: Tree Rational
cwTree = unfoldT (const False) frac left right (1, 1)
  where frac (m, n)  = m % n
        left (m, n)  = (m, m + n)
        right (m, n) = (n + m, n)
```

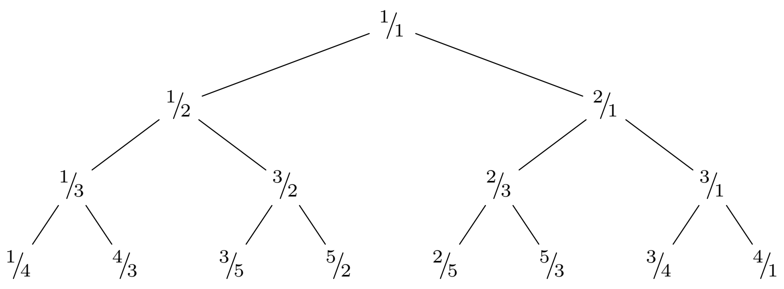

Fig. 1. The first few levels of the Calkin–Wilf tree

Here, *const a* is the function that always returns a, and the operator % constructs a rational from its numerator and denominator. For a full derivation of this algorithm, see [2, 49]; briefly, the paths in the tree correspond to traces of Euclid's algorithm computing the greatest common divisor of the numerator and denominator.

Another example of an unfold is given by the function *grow* that generates a binary search tree from a list of elements.

$$\begin{aligned}
&\mathit{grow} :: \mathit{Ord}\ a \Rightarrow \mathit{List}\ a \to \mathit{Tree}\ a\\
&\mathit{grow} = \mathit{unfoldT}\ \mathit{isNil}\ \mathit{head}\ \mathit{littles}\ \mathit{bigs}\\
&\mathit{littles}\ (\mathit{Cons}\ x\ \mathit{xs}) = \mathit{filterL}\ (\leqslant x)\ \mathit{xs}\\
&\mathit{bigs}\ (\mathit{Cons}\ x\ \mathit{xs}) \quad = \mathit{filterL}\ ({>}x)\ \mathit{xs}
\end{aligned}$$

(where *isNil* is the predicate that holds precisely of empty lists). As with the function *sort* mentioned in Section 2.4, *grow* has a type qualified by the context *Ord a*: the element type must be ordered.

3.4 Hylomorphisms

An unfold followed by a fold is a common pattern of computation [101]; 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.

$$\begin{aligned}
&\mathit{hyloL} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to c \to (a \to c \to c) \to b \to c\\
&\mathit{hyloL}\ p\ f\ g\ e\ h = \mathit{foldL}\ e\ h \circ \mathit{unfoldL}\ p\ f\ g\\
&\mathit{fact} :: \mathit{Integer} \to \mathit{Integer}\\
&\mathit{fact} = \mathit{hyloL}\ (0 ==)\ \mathit{id}\ \mathit{pred}\ 1\ (*)
\end{aligned}$$

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.

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

A similar definition can be given for binary trees, as shown below, together with an instance giving a kind of quicksort (albeit not a very quick one: it is not in-place, it has a bad space leak, and it takes quadratic time).

$$\begin{array}{l} \mathit{hyloT} :: (b \to \mathit{Bool}) \to (b \to a) \to (b \to b) \to (b \to b) \to \\ \qquad c \to (a \to c \to c \to c) \to b \to c \\ \mathit{hyloT}\; p\; f\; g_1\; g_2\; e\; h\; x \\ \quad = \mathbf{if}\; p\; x\; \mathbf{then}\; e \\ \qquad\quad \mathbf{else}\;\; h\; (f\; x)\; (\mathit{hyloT}\; p\; f\; g_1\; g_2\; e\; h\; (g_1\; x)) \\ \qquad\qquad\qquad\qquad (\mathit{hyloT}\; p\; f\; g_1\; g_2\; e\; h\; (g_2\; x)) \\ \mathit{qsort} :: \mathit{Ord}\; a \Rightarrow \mathit{List}\; a \to \mathit{List}\; a \\ \mathit{qsort} = \mathit{hyloT}\; \mathit{isNil}\; \mathit{head}\; \mathit{littles}\; \mathit{bigs}\; \mathit{Nil}\; \mathit{glue} \end{array}$$

3.5 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* [109, 110], *apomorphisms* [130] and *futumorphisms* [127], but these still all conform to the same structural scheme, and not all programs that generate data structures fit this scheme [46]. Gill et al. [50] 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. During the Spring School, Malcolm Wallace proposed the alternative term 'tectomorphism' for *build*, maintaining the Greek naming theme.

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 operator takes as argument a program with 'holes' for constructors, and plugs those holes with actual constructors.

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

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. (In fact, the argument g is the Church encoding of a list as a polymorphic lambda term, and *buildL* converts that encoding to a list as a familiar data structure [59].) We argued above that *unfoldL* is a dual to *foldL* in one sense; we make that sense clear in Section 3.8. 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:

```
foldL e f (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*:

```
unfoldL :: (b → Bool) → (b → a) → (b → b) → b → List a
unfoldL p f g b = buildL (h b)
  where h b n c = if p b then n else c (f b) (h (g b) n c)
```

However, some functions that generate lists can be expressed as an instance of *buildL* and not of *unfoldL* [46]; the well-known fast *reverse* is an example:

```
reverse xs = buildL (λn c → foldL id (λx g → g ∘ c x) xs n)
```

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 [37].

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
```

3.6 Datatype genericity

As we have already argued, data structure determines program structure [64]. It therefore makes sense to abstract from the determining shape, leaving only what programs of different shape have in common. What datatypes such as *List* and *Tree* have in common is the fact that they are recursive — which is to say, 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', but in this case with two arguments rather than one).

```
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)}
```

The generic datatype *Fix* is what the specific datatypes *List* and *Tree* have in common; the shape parameter *s* is what varies. Here are three instances of *Fix* using different shapes: lists and internally labelled binary trees as seen before, and also a datatype of 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 the types *List* and *Tree* here are equivalent to but different from the types *List* in Section 3.1 and *Tree* in Section 3.3.

The datatype *Fix s a* is a recursive type; the type constructor *Fix* ties the recursive knot around the shape *s*. Typically, as in the three instances above, the shape *s* has several variants, including a 'base case' independent of the second argument. But with lazy evaluation, infinite structures are possible, and so the definition makes sense even with no base case. For example, the datatype *Fix ITreeF a* with shape parameter **data** *ITreeF a b* = *INodeF a b b* is a type of infinite internally labelled binary trees, which would suffice for the *cwTree* example above.

3.7 Bifunctors

Not all valid binary type constructors *s* are suitable for *Fix*ing; for example, function types with the parameter appearing in *contravariant* (source) positions cause problems. It turns out that we should restrict attention to (covariant) *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:

$$\begin{aligned} bimap\ id\ id &= id \\ bimap\ f\ g \circ bimap\ h\ j &= bimap\ (f \circ h)\ (g \circ j) \end{aligned}$$

These properties cannot be expressed formally in most languages today, as we noted in Section 2.5, but we might expect to be able to express them in the languages of tomorrow [18, 115], and they are important for reasoning about programs using *bimap*.

All sum-of-product datatypes — that is, consisting of a number of variants, each with a number of arguments — 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 TreeF where
  bimap f g EmptyF        = EmptyF
  bimap f g (NodeF x y z) = NodeF (f x) (g y) (g z)
instance Bifunctor BtreeF where
  bimap f g (TipF x)   = TipF (f x)
  bimap f g (BinF y z) = BinF (g y) (g z)
```

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

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

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 [60] is designed, and one can almost achieve the same effect in Haskell [17, 58, 26]. One might hope that these instance definitions would in fact be inferred, in the languages of tomorrow [97, 62]. 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.

3.8 Datatype-generic recursion patterns

It turns out that the class *Bifunctor* provides sufficient flexibility to capture a wide variety of recursion patterns as datatype-generic programs. The datatype-specific recursion patterns introduced above can all be made generic in a bifunctorial shape s; a little bit of ad-hockery goes a long way. (These definitions are very similar to those in the PolyP approach [68], discussed in more detail in [61, §4.2] in this volume.)

```
map :: Bifunctor s ⇒ (a → b) → (Fix s a → Fix s b)
map f = In ∘ bimap f (map f) ∘ out
fold ::   Bifunctor s ⇒ (s a b → b) → Fix s a → b
fold f = f ∘ bimap id (fold f) ∘ out
unfold :: Bifunctor s ⇒ (b → s a b) → b → Fix s a
unfold f = In ∘ bimap id (unfold f) ∘ f
hylo ::   Bifunctor s ⇒ (b → s a b) → (s a c → c) → b → c
hylo f g = g ∘ bimap id (hylo f g) ∘ f
build ::  Bifunctor s ⇒ (∀b. (s a b → b) → b) → Fix s a
build f = f In
```

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 promised duality between *fold* and *unfold* is readily apparent. (But note that these datatype-generic definitions are applicable only to instantiations of *Fix*, as in Section 3.6, and not to the datatypes of the same name in Section 2.)

4 The Origami patterns

Design patterns, as the subtitle of the seminal book [35] has it, are 'elements of reusable object-oriented software'. However, within the confines of existing mainstream 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 those mainstream programming languages. Specifically, we argue that what those languages lack are higher-order and datatype-generic features; given such features, the code parts of some design patterns at least are expressible as directly reusable library components. The benefits of expressing patterns in this way are considerable: patterns may then be reasoned about, type-checked, applied and reused, just as any other abstraction can.

We argue our case by capturing as higher-order datatype-generic programs a small subset ORIGAMI of the Gang of Four (GOF) patterns. (Within these notes, for simplicity, we equate GOF patterns with design patterns; we use SMALL CAPITALS for the names of 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 mainstream languages of today — strongly and statically typed, object-oriented, with an underlying imperative approach — but incorporating additional features from the functional world — specifically, higher-order operators and datatype genericity.

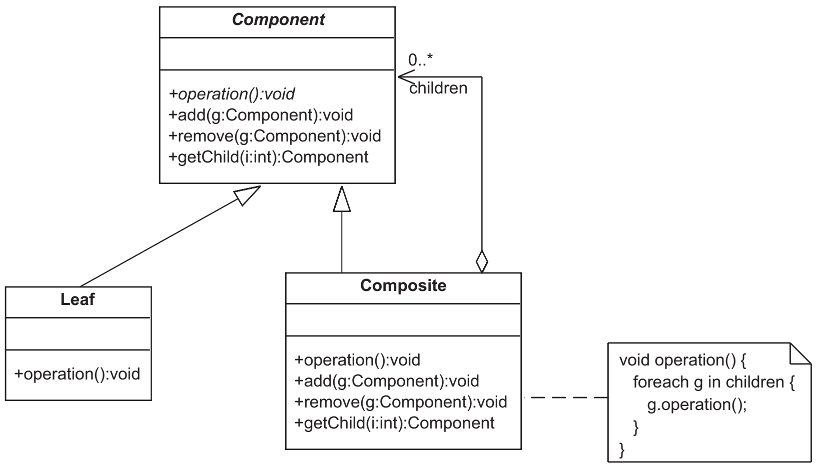

Fig. 2. The class structure of the COMPOSITE pattern

4.1 The Origami family of patterns

In this section we describe ORIGAMI, a little suite of patterns for recursive data structures, consisting of four of the Gang of Four design patterns [35]:

- 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.

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 (*Component*), of which both atomic (*Leaf*) and aggregate (*Composite*) objects are subtypes, as shown in Figure 2.

Iterator The ITERATOR pattern 'provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation'.

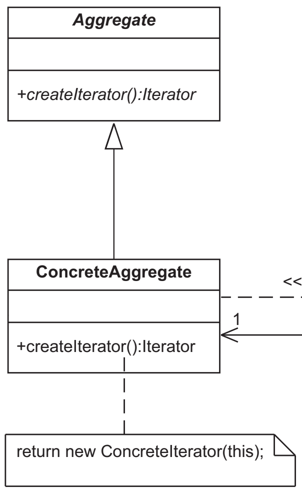

Fig. 3. The class structure of the EXTERNAL ITERATOR pattern

It does this by separating the responsibilities of containment (*Aggregate*) and iteration (*Iterator*). The standard implementation is as an *external* or client-driven iterator, illustrated in Figure 3 and as embodied for example in the Java standard library.

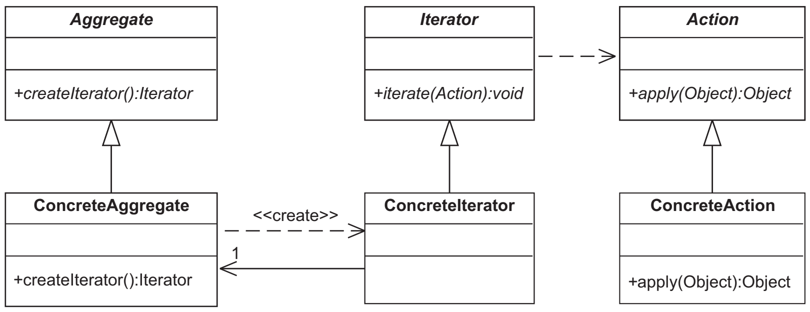

Fig. 4. The class structure of the INTERNAL ITERATOR pattern

In addition to the standard implementation, GOF also discuss *internal* or iterator-driven ITERATORS, illustrated in Figure 4. These might be modelled by the following pair of Java interfaces:

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

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

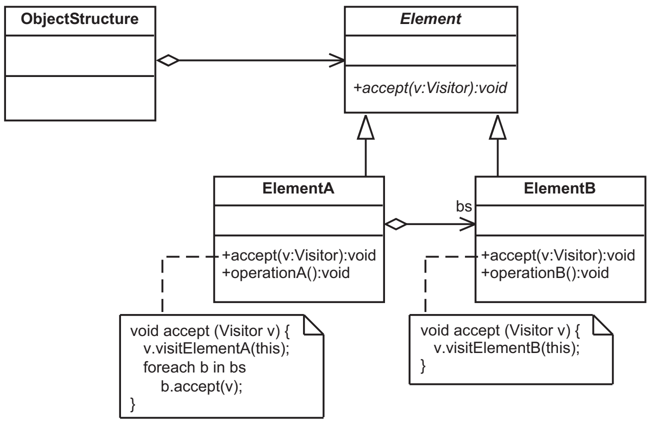

Fig. 5. The class structure of the elements in the INTERNAL VISITOR pattern

modified) element. The C++ STL calls such objects 'functors', but we avoid that term here to prevent a name clash with type functors. A collection (implements a FACTORY METHOD [35] 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.

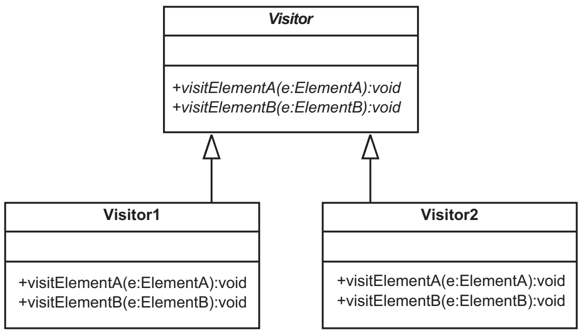
Visitor
+visitElementA(e:ElementA):void
+visitElementB(e:ElementB):void
Visitor1
+visitElementA(e:ElementA):void
+visitElementB(e:ElementB):void
Visitor2
+visitElementA(e:ElementA):void
+visitElementB(e:ElementB):void

Fig. 6. The class structure of the visitors in the INTERNAL VISITOR pattern

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

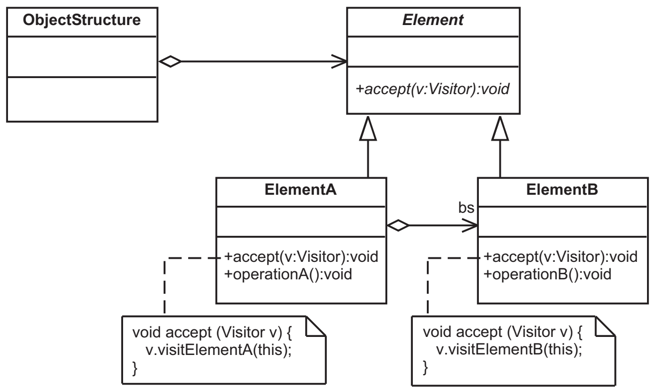

Fig. 7. The class structure of the elements in the External Visitor pattern

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 5), and an interface for those traversals to implement (the interface *Visitor* in Figure 6); the effect is to simulate *double dispatch* on the types of two arguments, the element type and the operation, by two consecutive single dispatches.

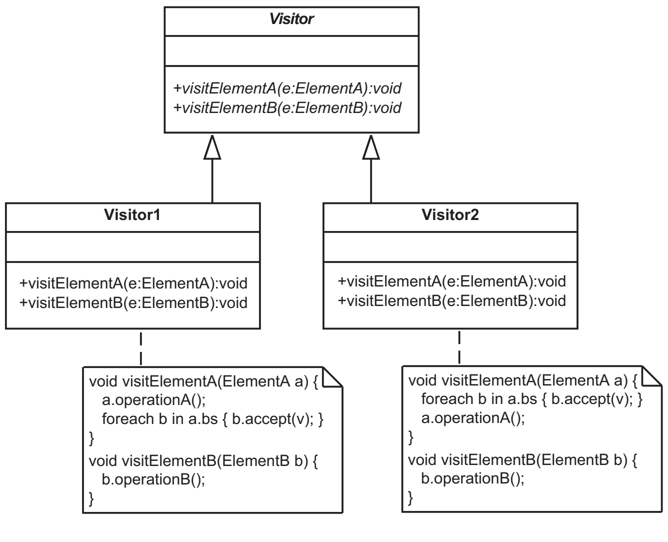
Visitor
+visitElementA(e:ElementA):void
+visitElementB(e:ElementB):void
Visitor1
+visitElementA(e:ElementA):void
+visitElementB(e:ElementB):void
Visitor2
+visitElementA(e:ElementA):void
+visitElementB(e:ElementB):void
void visitElementA(ElementA a) {
a.operationA();
foreach b in a.bs { b.accept(v); }
}
void visitElementB(ElementB b) {
b.operationB();
}
void visitElementA(ElementA a) {
foreach b in a.bs { b.accept(v); }
a.operationA();
}
void visitElementB(ElementB b) {
b.operationB();
}

Fig. 8. The class structure of the visitors in the External Visitor pattern

The pattern provides a kind of *aspect-oriented programming* [82], modularizing what would otherwise be a cross-cutting concern, namely the definition of a traversal. It reverses the costs: it is now easy to add new traversals, but hard to add new variants. (Wadler [137] has coined the term *expression problem* for this tension between dimensions of easy extension.)

As with the distinction between internal and external iterators, there is a choice about where to put responsibility for managing a traversal. Buchlovsky and Thielecke [15] use the term 'INTERNAL VISITOR' for the usual presentation, with the *accept* methods of *Element* subclasses making recursive calls as shown in Figure 5. Moving that responsibility from the *accept* methods of the *Element* classes to the *visit* methods of the *Visitor* classes, as shown in Figures 7 and 8, yields what they call an EXTERNAL VISITOR. Now the traversal algorithm is not fixed, and different visitors may vary it (for example, between preorder and postorder). One might say that this latter variation encapsulates simple case analysis or pattern matching, rather than traversals per se.

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 9 shows, this is done by delegating responsibility for the construction to a separate *Builder* object — in fact, an instance of the STRATEGY pattern [35], encapsulating a strategy for performing the construction.

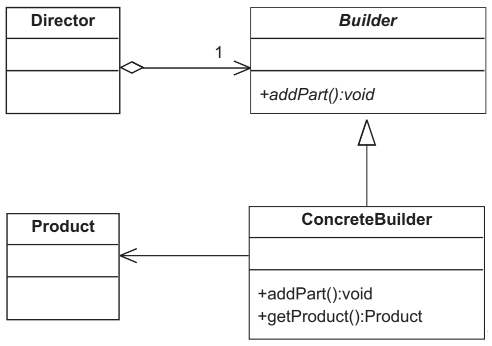

Fig. 9. The class structure of the BUILDER pattern

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 take just a part and return no result. 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; a directed acyclic graph requires a set of parents for each node, and construction in topological order; a cyclic graph requires the possibility of 'forward references', adding parts as children of yet-to-be-added parents.

GOF also suggest the possibility of COMPUTING BUILDERS. 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.

4.2 An application of Origami

As an example of applying the ORIGAMI patterns, consider the little document system illustrated in Figure 10. (The code for this example is presented as an appendix in Section 7.)

- The focus of the application is the 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* titles 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 change the identity of the document elements; more generally, *apply* could optionally return new elements, as described under the ITERATOR pattern above.)
- 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 of the pattern, adding *Section*s and *Paragraph*s as children of existing *Component*s via unique *int* identifiers (only non-negative *int*s are returned as identifiers, so a parentless node can be indicated by passing a negative *int*). A *ComponentBuilder* constructs a *Component* as expected, whereas a *PrintBuilder* is a COMPUTING BUILDER, incorporating the printing behaviour of the *PrintVisitor* incrementally and 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 4.3 below, we provide just such a representation, in terms of the higher-order datatype-generic programs from Section 3.8. Section 4.4 justifies our claim of a faithful rendition by capturing the structure of the document application in this alternative representation.

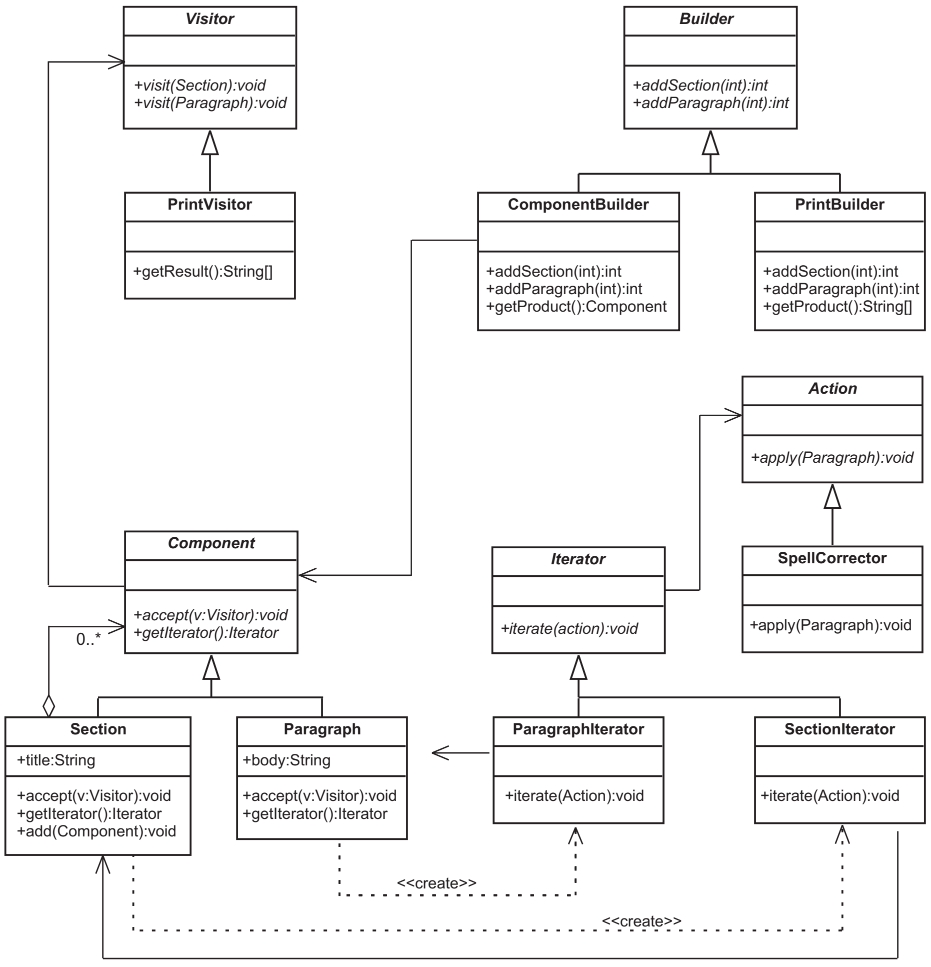

Fig. 10. An application of the Origami patterns

4.3 Patterns as HODGPs

We now revisit the ORIGAMI patterns, showing that each of the four patterns can be captured using higher-order datatype-generic program (HODGP) constructs. 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.

Composite in HODGP COMPOSITEs are just recursive data structures. So actually, these correspond not 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* (technically *regular*, that is, first-order fixed points of type functors admitting a *map* operation) recursive data structures, whatever their shape.

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. What was a double dispatch in the OO setting becomes in HODGP the choice of a function to apply, followed by a case analysis on the variant of the data structure. A common case of such traversals, albeit not the most general, is the fold operator introduced above.

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

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

Iterator in HODGP EXTERNAL ITERATORs give sequential access to the elements of a collection. The functional approach would be to provide a view of the collection as a list of elements, at least for read-only access. Seen this way, the ITERATOR pattern can be implemented using the VISITOR pattern, traversing using a body *combiner* that combines the element lists from substructures into one overall element list.

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

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' [136].

In the formulation above, the *combiner* argument has to be provided to the *elements* 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 operator. Thus, element consumers can already be written datatype-generically today. 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; it therefore supports write access to the collection as well as read access. In HODGP, we can give a *single generic* definition of this.

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

This is in contrast with the object-oriented approach, in which internal *Iterator* implementations are ad-hoc datatype-generic. Note also that the HODGP version is more general than the OO version, because it can safely return a collection of elements of a different type.

On the other hand, the object-oriented ITERATOR can have side-effects, which the purely functional *map* cannot; for example, it can perform I/O, accumulate a measure of the collection, and so on. However, it is possible to generalize the *map* operation considerably, capturing all those effects in a datatype-generic way. This is the subject of Section 5.

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.

$$\begin{array}{l}\mathit{unfold} \;\; :: \mathit{Bifunctor}\ s \Rightarrow (b \to s\ a\ b) \to b \to \mathit{Fix}\ s\ a \\ \mathit{unfold}\ f = \mathit{In} \circ \mathit{bimap}\ \mathit{id}\ (\mathit{unfold}\ f) \circ f\end{array}$$

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

$$\begin{array}{l}\mathit{build} \;\; :: \mathit{Bifunctor}\ s \Rightarrow (\forall b.\ \ (s\ a\ b \to b) \to b) \to \mathit{Fix}\ s\ a \\ \mathit{build}\ f = f\ \mathit{In}\end{array}$$

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 at once.

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 [133], 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.

$$\begin{array}{l}\mathit{hylo} \;\; :: \mathit{Bifunctor}\ s \Rightarrow (b \to s\ a\ b) \to (s\ a\ c \to c) \to b \to c \\ \mathit{hylo}\ f\ g = g \circ \mathit{bimap}\ \mathit{id}\ (\mathit{hylo}\ f\ g) \circ f\end{array}$$

More generally, but harder to reason with, the producer is a build, and the composition replaces the constructors in the builder by the body of the fold.

$$\begin{array}{l}\mathit{foldBuild} \;\; :: \mathit{Bifunctor}\ s \Rightarrow (\forall b.\ \ (s\ a\ b \to b) \to b) \to (s\ a\ c \to c) \to c \\ \mathit{foldBuild}\ f\ g = f\ g\end{array}$$

(that is, $\mathit{foldBuild}\ f\ g = \mathit{fold}\ g\ (\mathit{build}\ f)$.) Once again, both of these definitions are datatype-generic; both take 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 — see the code for *PrintBuilder* in Section 7.14.

4.4 The example, revisited

To justify our claim that the higher-order datatype-generic representation of the Origami patterns is a faithful rendition, we use it to re-express the document application discussed in Section 4.2 and illustrated in Figure 10. It is instructive to compare these 40 lines of Haskell code with the equivalent Java code in Section 7.

- The COMPOSITE structure has the following shape.

```
data DocF a b = Para a | Sec String [b]
type Doc = Fix DocF String
instance Bifunctor DocF where
  bimap f g (Para s)   = Para (f s)
  bimap f g (Sec s xs) = Sec s (map g xs)
```

  We have chosen to consider paragraph bodies as the 'contents' of the data structure, but section titles as part of the 'shape'; then mapping over the contents will affect the paragraph bodies but not the section titles. That decision could easily be varied.
- We used an INTERNAL ITERATOR to implement the *SpellCorrector*; this would be modelled now as an instance of *map*.

```
correct :: String → String   -- definition omitted
corrector :: Doc → Doc
corrector = map correct
```

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

```
printDoc :: Doc → [String]
printDoc = fold combine
combine :: DocF String [String] → [String]
combine (Para s)   = [s]
combine (Sec s xs) = s : concat xs
```

- 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 trees.

```
data XML = Text String | Entity Tag Attrs [XML]
type Tag   = String
type Attrs = [(String, String)]
```

  From such an XML tree we can construct a document.

```
fromXML :: XML → Doc
fromXML = unfold element
```

  Text elements are represented as paragraphs, and *Entity*s as sections having appropriate titles.

```
element :: XML → DocF String XML
element (Text s)            = Para s
element (Entity t kvs xs) = Sec (title t kvs) xs
```

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

- Printing of a document constructed from an XML file is the composition of a fold with an unfold.

  $$
  \begin{aligned}
  &\mathit{printXML} :: \mathit{XML} \to [\mathit{String}]\\
  &\mathit{printXML} = \mathit{printDoc} \circ \mathit{fromXML}
  \end{aligned}
  $$

  It is therefore also a hylomorphism:

  $$\mathit{printXML} = \mathit{hylo}\ \mathit{element}\ \mathit{combine}$$

- 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.

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

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

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

  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.

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

5 The Essence of the Iterator pattern

In Section 4, we argued that the ITERATOR pattern amounts to nothing more than the higher-order datatype-generic *map* operation. However, as we mentioned, there are aspects of an ITERATOR that are not adequately explained by a *map*; in particular, the possibility of effects such as I/O, and dependencies between the actions executed at each element.

For example, consider the code below, showing a C# method *loop* that iterates over a collection, counting the elements and simultaneously interacting with each of them.

```
public static int loop⟨MyObj⟩ (IEnumerable⟨MyObj⟩ coll){
  int n = 0;
  foreach (MyObj obj in coll){
    n = n + 1;
    obj.touch ();
  }
  return n;
}
```

The method is parametrized by the type *MyObj* of collection elements; this parameter is used twice: to constrain the collection *coll* passed as a parameter, and as a type for the local variable *obj*. The collection itself is rather unconstrained; it only has to implement the *IEnumerable*⟨*MyObj*⟩ interface.

In this section, we investigate the structure of such iterations. We emphasize that we want to capture both aspects of the method *loop* and iterations like it: *mapping* over the elements, and simultaneously *accumulating* some measure of those elements. This takes us beyond the more simplistic *map* model from Section 4. We still aim to construct a *holistic* model, treating the iteration as an abstraction in its own right; this leads us naturally to a higher-order presentation. We also want to develop an *algebra* of such iterations, with combinators for composing them and laws for reasoning about them; this strengthens the case for a declarative approach. We argue that McBride and Paterson's recently introduced notion of *idioms* [95], and in particular the corresponding *traverse* operator, have exactly the right properties. (The material in this section is based on [44]; the reader is warned that this work, and especially the results in Section 5.7, is more advanced and less mature than in the earlier parts of these notes.)

5.1 Functional iteration

In this section, we review a number of simpler approaches to capturing the essence of iteration. In particular, we look at a variety of datatype-generic recursion operators: maps, folds, unfolds, crushes, and monadic maps. The traversals we discuss in Section 5.3 generalize all of these.

Origami We have already considered the *origami* style of programming [37, 38], in which the structure of programs is captured by higher-order datatype-generic recursion operators such as *map*, *fold* and *unfold*. And we have already observed that the recursion pattern *map* captures iterations that modify each element of a collection independently; thus, *map touch* captures the mapping aspect of the C# loop above, but not the accumulating aspect.

At first glance, it might seem that the datatype-generic *fold* captures the accumulating aspect; but the analogy is rather less clear for a non-linear collection. In contrast to the C# program above, which is sufficiently generic to apply to non-linear collections, a datatype-generic counting operation defined using *fold* would need a datatype-generic numeric algebra as the fold body. Such a thing could be defined polytypically [68, 60], but the fact remains that *fold* in isolation does not encapsulate the datatype genericity.

Essential to iteration in the sense we are using the term is linear access to collection elements; this was the problem with *fold*. One might consider a datatype-generic operation to yield a linear sequence of collection elements from possibly non-linear structures, for example by *fold*ing with a content combiner, or *unfold*ing to a list. This could be done (though as with the *fold* problem, it requires a datatype-generic sequence algebra or coalgebra as the body of the *fold* or *unfold*); but even then, this would address only the accumulating aspect of the C# iteration, and not the mapping aspect — it loses the shape of the original structure. Moreover, although the sequence of elements is always definable as an instance of *fold*, it is not always definable as an instance of *unfold* [46].

We might also explore the possibility of combining some of these approaches. For example, it is clear from the definitions above that *map* is an instance of *fold*. Moreover, the *banana split theorem* [31] states that two folds in parallel on the same data structure can be fused into one. Therefore, a map and a fold in parallel fuse to a single fold, yielding both a new collection and an accumulated measure, and might therefore be considered to capture both aspects of the C# iteration. However, we feel that this is an unsatisfactory solution: it may indeed simulate or implement the same behaviour, but it is no longer manifest that the shape of the resulting collection is related to that of the original.

Crush Meertens [99] generalized APL's 'reduce' [67] to a *crush* operation, $\langle\!\langle \oplus \rangle\!\rangle :: t\ a \rightarrow a$ for binary operator $(\oplus) :: a \rightarrow a \rightarrow a$ with a unit, polytypically over the structure of a regular functor t. For example, $\langle\!\langle + \rangle\!\rangle$ polytypically sums a collection of numbers. For projections, composition, sum and fixpoint, there is an obvious thing to do, so the only ingredients that need to be provided are the binary operator (for products) and a constant (for units). Crush captures the accumulating aspect of the C# iteration above, accumulating elements independently of the shape of the data structure, but not the mapping aspect.

Monadic map Haskell's standard library [112] defines a *monadic map* for lists, which lifts the standard map on lists (taking a function on elements to a function on lists) to the Kleisli category (taking a monadic function on elements to a monadic function on lists):

$$mapM :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow ([a] \rightarrow m\ [b])$$

(For notational brevity, we resort throughout Section 5 to the built-in type $[a]$ of lists rather than the datatype-generic type *List* a.) Fokkinga [30] showed how to generalize this from lists to an arbitrary regular functor, datatype-generically. Several authors [102, 106, 70, 110, 84] have observed that monadic map is a promising model of iteration. Monadic maps are very close to the *idiomatic traversals* that we propose as the essence of imperative iterations; indeed, for certain idioms — specifically, those that arise from monads — traversal reduces exactly to monadic map. However, we argue that monadic maps do not capture accumulating iterations as nicely as they might. Moreover, it is well-known [77, 83] that monads do not compose in general, whereas it turns out that idioms do; this gives us a richer algebra of traversals. Finally, monadic maps stumble over products, for which there are two reasonable but symmetric definitions, coinciding when the monad is commutative. This stumbling block forces either a bias to left or right, or a restricted focus on commutative monads, or an additional complicating parametrization; in contrast, idioms generally have no such problem, and in fact turn it into a virtue.

Closely related to monadic maps are operations like Haskell's *sequence* function:

$$sequence :: Monad\ m \Rightarrow [m\ a] \rightarrow m\ [a]$$

and its datatype-generic generalization to arbitrary datatypes. Indeed, *sequence* and *mapM* are interdefinable:

$$mapM\ f = sequence \circ map\ f$$

and so

$$sequence = mapM\ id$$

Most writers on monadic maps have investigated such an operation; Moggi *et al.* [106] call it *passive traversal*, Meertens [100] calls it *functor pulling*, and Pardo [110] and others have called it a *distributive law*. It is related to Hoogendijk and Backhouse's *commuting relators* [65], but with the addition of the monadic structure on one of the functors. McBride and Paterson introduce the function *dist* playing the same role, but as we shall see, more generally.

5.2 Idioms

McBride and Paterson [95] recently introduced the notion of an *idiom* or *applicative functor* as a generalization of monads. ('Idiom' was the name McBride originally chose, but he and Paterson now favour the less evocative term 'applicative functor'. We prefer the original term, not least because it lends itself nicely to adjectival uses, as in 'idiomatic traversal'.) Monads [105, 135] allow the expression of effectful computations within a purely functional language, but they do so by encouraging an *imperative* [114] programming style; in fact, Haskell's monadic **do** notation is explicitly designed to give an imperative feel. Since idioms generalize monads, they provide the same access to effectful computations; but they encourage a more *applicative* programming style, and so fit better within the functional programming milieu. Moreover, as we shall see, idioms strictly generalize monads; they provide features beyond those of monads. This will be important to us in capturing a wider variety of iterations, and in providing a richer algebra of those iterations.

Idioms are captured in Haskell by the following type class. (In contrast to McBride and Paterson, we insist that every *Idiom* is also a *Functor*. This entails no loss of generality, since the laws below ensure that defining $\mathit{fmap}\ f\ x = \mathit{pure}\ f \circledast x$ suffices.)

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Functor}\ m \Rightarrow \mathit{Idiom}\ m\ \mathbf{where} \\
\quad \mathit{pure} :: a \rightarrow m\ a \\
\quad (\circledast)\ \ :: m\ (a \rightarrow b) \rightarrow (m\ a \rightarrow m\ b)
\end{array}
$$

Informally, *pure* lifts ordinary values into the idiomatic world, and $\circledast$ provides an idiomatic flavour of function application. We make the convention that $\circledast$ associates to the left, just like ordinary function application.

In addition to those inherited from the *Functor* class, idioms are expected to satisfy the following laws.

$$
\begin{array}{ll}
\mathit{pure}\ \mathit{id} \circledast u & = u \\
\mathit{pure}\ (\circ) \circledast u \circledast v \circledast w & = u \circledast (v \circledast w) \\
\mathit{pure}\ f \circledast \mathit{pure}\ x & = \mathit{pure}\ (f\ x) \\
u \circledast \mathit{pure}\ x & = \mathit{pure}\ (\lambda f \rightarrow f\ x) \circledast u
\end{array}
$$

(recall that $(\circ)$ denotes function composition). These two collections of laws are together sufficient to allow any expression built from the idiom operators to be rewritten into a canonical form, consisting of a pure function applied to a series of idiomatic arguments:

$$\mathit{pure}\ f \circledast u_1 \circledast \ldots \circledast u_n$$

(In case the reader feels the need for some intuition for these laws, we refer them forwards to the stream Naperian idiom discussed below.)

Monadic idioms Idioms generalize monads; every monad induces an idiom, with the following operations. (Taken literally as a Haskell declaration, this code provokes complaints about overlapping instances; it is therefore perhaps better to take it as a statement of intent instead.)

$$
\begin{array}{l}
\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Idiom}\ m\ \mathbf{where} \\
\quad \mathit{pure}\ a \quad\ \ = \mathbf{do}\ \{\mathit{return}\ a\} \\
\quad \mathit{mf} \circledast \mathit{mx} = \mathbf{do}\ \{f \leftarrow \mathit{mf}; x \leftarrow \mathit{mx}; \mathit{return}\ (f\ x)\}
\end{array}
$$

The *pure* operator for a monadic idiom is just the *return* of the monad; idiomatic application $\circledast$ is monadic application, here with the effects of the function preceding those of the argument. There is another, completely symmetric, definition, with the effects of the argument before those of the function. We leave it to the reader to verify that the monad laws entail the idiom laws (with either definition of monadic application).

Naperian idioms One of McBride and Paterson's motivating examples of an idiom arises from the environment monad:

```
newtype Env e a = Env{ unEnv :: e → a}
```

The *pure* and ⊛ of this type turn out to be the K and S combinators, respectively.

```
instance Idiom (Env e) where
  pure a              = Env (λe → a)
  Env ef ⊛ Env ex = Env (λe → (ef e) (ex e))
```

One can think of instances of *Env e* as datatypes with *fixed shape*, which gives rise to an interesting subclass of monadic idioms. For example, the functor *Stream* is equivalent to *Env Nat*; under the equivalence, the K and S combinators turn out to be the familiar 'repeat' and 'zip with apply' operators.

```
data Stream a = ConsS a (Stream a)
instance Idiom Stream where
  pure a     = repeatS a
  mf ⊛ mx = zipApS mf mx
repeatS   :: a → Stream a
repeatS x = xs where xs = ConsS x xs
zipApS :: (a → b → c) → Stream a → Stream b → Stream c
zipApS (ConsS f fs) (ConsS x xs) = ConsS (f x) (zipApS fs xs)
```

The *pure* operator lifts a value to a stream, replicating it for each element; idiomatic application is pointwise, taking a stream of functions and a stream of arguments to a stream of results. We find that this idiom is the most accessible one for understanding the idiom laws.

A similar construction works for any fixed-shape datatype: pairs, vectors of length n, two-dimensional matrices of a given size, infinite binary trees, and so on. Peter Hancock [94] calls such a datatype *Naperian*, because the environment or position type acts as a notion of logarithm. That is, datatype t is Naperian if $t\ a \simeq a^p \simeq p \to a$ for some type p of positions, called the logarithm $\log\ t$ of t. Then $t\ 1 \simeq 1^p \simeq 1$, so the shape is fixed, and familiar properties of logarithms arise — for example, $\log\ (t \circ u) \simeq \log\ t \times \log\ u$.

```
class Functor t ⇒ Naperian t p | p → t,  t → p where
  fill :: (p → a) → t a        -- index ∘ fill = id
  index :: t a → (p → a)    -- fill ∘ index = id
  indices :: t p
  indices = fill id
  fill f = fmap f indices
```

(Here, $p \to t, t \to p$ are *functional dependencies*, indicating that each of the two parameters of the type class *Naperian* determines the other.) We leave as an exercise for the reader to show that the definitions

$$\begin{aligned} &\mathit{pure}\ a &&= \mathit{fill}\ (\lambda p \rightarrow a) \\ &\mathit{mf} \circledast \mathit{mx} &&= \mathit{fill}\ (\lambda p \rightarrow (\mathit{index}\ \mathit{mf}\ p)\ (\mathit{index}\ \mathit{mx}\ p)) \end{aligned}$$

satisfy the idiom laws.

Naperian idioms impose a fixed and hence statically known shape on data. We therefore expect some connection with data-parallel and numerically intensive computation, in the style of Jay's language FISh [71] and its *shapely operations* [72], which separate statically analysable shape from dynamically determined contents. Computations within Naperian idioms tend to perform a transposition of results; there appears also to be some connection with what Kühne [86] calls the *transfold* operator.

The 'bind' operation of a monad allows the result of one computation to affect the choice and ordering of effects of subsequent operations. Idioms in general provide no such possibility; indeed, as we have seen, every expression built just from the idiom combinators is equivalent to a pure function applied to a series of idiomatic arguments, and so the sequencing of any effects is fixed. Focusing on the idiomatic view of a Naperian datatype, rather than the monadic view in terms of an environment, enforces that restriction. The interesting thing is that many useful computations involving monads do not require the full flexibility of dynamically chosen ordering of effects; for these, the idiomatic interface suffices.

Monoidal idioms Idioms strictly generalize monads; there are idioms that do not arise from monads. A third family of idioms, this time non-monadic, arises from constant functors with monoidal targets. McBride and Paterson call these *phantom idioms*, because the resulting type is a phantom type (as opposed to a container type of some kind). Any monoid $(\emptyset, \oplus)$ induces an idiom, where the *pure* operator yields the unit of the monoid and application uses the binary operator.

```
newtype K b a = K{unK :: b}
instance Monoid b ⇒ Idiom (K b) where
  pure _ = K ∅
  x ⊛ y  = K (unK x ⊕ unK y)
```

Computations within this idiom accumulate some measure: for the monoid of integers with addition, they count or sum; for the monoid of lists with concatenation, they collect some trace of values; for the monoid of booleans with disjunction, they encapsulate linear searches; and so on. Note that sequences of one kind or another therefore form idioms in three different ways: monadic with cartesian product, modelling non-determinism; Naperian with zip, modelling data-parallelism; and monoidal with concatenation, modelling tracing.

Combining idioms Like monads, idioms are closed under products; so two independent idiomatic effects can generally be fused into one, their product.

```
data Prod m n a = Prod{pfst :: m a, psnd :: n a}
```

```
fork :: (a → m b) → (a → n b) → a → Prod m n b
fork f g a = Prod (f a) (g a)
instance (Idiom m, Idiom n) ⇒ Idiom (Prod m n) where
  pure x    = Prod (pure x) (pure x)
  mf ⊛ mx = Prod (pfst mf ⊛ pfst mx) (psnd mf ⊛ psnd mx)
```

Unlike monads in general, idioms are also closed under composition; so two sequentially dependent idiomatic effects can generally be fused into one, their composition.

```
data Comp m n a = Comp{ unComp :: m (n a)}
instance (Idiom m, Idiom n) ⇒ Idiom (Comp m n) where
  pure x    = Comp (pure (pure x))
  mf ⊛ mx = Comp (pure (⊛) ⊛ unComp mf ⊛ unComp mx)
```

We see examples of both of these combinations in Section 5.4.

5.3 Idiomatic traversal

Two of the three motivating examples McBride and Paterson provide for idiomatic computations, sequencing a list of monadic effects and transposing a matrix, are instances of a general scheme they call *traversal*. This involves iterating over the elements of a data structure, in the style of a 'map', but interpreting certain function applications within the idiom.

In the case of lists, traversal may be defined as follows.

```
traverseL :: Idiom m ⇒ (a → m b) → ([a] → m [b])
traverseL f [ ]      = pure [ ]
traverseL f (x : xs) = pure (:) ⊛ f x ⊛ traverseL f xs
```

A special case is for the identity function, when traversal distributes the data structure over the idiomatic structure:

```
distL :: Idiom m ⇒ [m a] → m [a]
distL = traverseL id
```

The 'map within the idiom' pattern of traversal for lists generalizes to any (finite) functorial data structure. We capture this via a type class of *Traversable* data structures (again, unlike McBride and Paterson, but without loss of generality, we insist on functoriality):

```
class Functor t ⇒ Traversable t where
  traverse   :: Idiom m ⇒ (a → m b) → (t a → m (t b))
  dist       :: Idiom m ⇒ t (m a) → m (t a)
  traverse f = dist ∘ fmap f
  dist       = traverse id
```

As intended, this class generalizes *traverseL*:

$$\textbf{instance}\ \mathit{Traversable}\ [\,]\ \textbf{where}\ \mathit{traverse} = \mathit{traverseL}$$

Although *traverse* and *dist* are interdefinable (intuitively, *dist* is to *traverse* as monadic join μ is to bind $\gg\!=$), so only one needs to be given, defining *traverse* and inheriting *dist* is usually simpler and more efficient than vice versa.

$$\begin{array}{l}
\textbf{data}\ \mathit{Btree}\ a = \mathit{Tip}\ a \mid \mathit{Bin}\ (\mathit{Btree}\ a)\ (\mathit{Btree}\ a)\\
\textbf{instance}\ \mathit{Traversable}\ \mathit{Btree}\ \textbf{where}\\
\quad \mathit{traverse}\ f\ (\mathit{Tip}\ a)\quad\ \ = \mathit{pure}\ \mathit{Tip} \circledast f\ a\\
\quad \mathit{traverse}\ f\ (\mathit{Bin}\ t\ u) = \mathit{pure}\ \mathit{Bin} \circledast \mathit{traverse}\ f\ t \circledast \mathit{traverse}\ f\ u
\end{array}$$

McBride and Paterson propose a special syntax involving 'idiomatic brackets', which would have the effect of inserting the occurrences of *pure* and $\circledast$ implicitly; apart from these brackets, the definition of *traverse* then looks exactly like a definition of *fmap*. This definition could be derived automatically [62], or given polytypically once and for all, assuming some universal representation of datatypes such as sums and products [60] or regular functors [38]:

$$\begin{array}{l}
\textbf{class}\ \mathit{Bifunctor}\ s \Rightarrow \mathit{Bitraversable}\ s\ \textbf{where}\\
\quad \mathit{bidist} :: \mathit{Idiom}\ m \Rightarrow s\ (m\ a)\ (m\ b) \to m\ (s\ a\ b)\\
\textbf{instance}\ \mathit{Bitraversable}\ s \Rightarrow \mathit{Traversable}\ (\mathit{Fix}\ s)\ \textbf{where}\\
\quad \mathit{traverse}\ f = \mathit{fold}\ (\mathit{fmap}\ \mathit{In} \circ \mathit{bidist} \circ \mathit{bimap}\ f\ \mathit{id})\\
\textbf{instance}\ \mathit{Bitraversable}\ \mathit{BtreeF}\ \textbf{where}\\
\quad \mathit{bidist}\ (\mathit{TipF}\ a)\quad\ \ = \mathit{pure}\ \mathit{TipF} \circledast a\\
\quad \mathit{bidist}\ (\mathit{BinF}\ t\ u) = \mathit{pure}\ \mathit{BinF} \circledast t \circledast u
\end{array}$$

When m is specialized to the identity idiom, traversal reduces to the functorial map over lists.

$$\begin{array}{l}
\textbf{newtype}\ \mathit{Id}\ a = \mathit{Id}\{\mathit{unId} :: a\}\\
\textbf{instance}\ \mathit{Idiom}\ \mathit{Id}\ \textbf{where}\\
\quad \mathit{pure}\ a\quad\ \ = \mathit{Id}\ a\\
\quad \mathit{mf} \circledast \mathit{mx} = \mathit{Id}\ ((\mathit{unId}\ \mathit{mf})\ (\mathit{unId}\ \mathit{mx}))
\end{array}$$

In the case of a monadic idiom, traversal specializes to monadic map, and has the same uses. In fact, traversal is really just a slight generalization of monadic map: generalizing in the sense that it applies also to non-monadic idioms. We consider this an interesting insight, because it reveals that monadic-map-like traversal in some sense does not require the full power of a monad; in particular, it does not require the bind or join operators, which are unavailable in idioms in general.

For a Naperian idiom, traversal transposes results. For example, interpreted in the pair Naperian idiom, *traverseL id* unzips a list of pairs into a pair of lists.

For a monoidal idiom, traversal accumulates values. For example, interpreted in the integer monoidal idiom, traversal accumulates a sum of integer measures of the elements.

$$\begin{array}{l} \mathit{tsum} :: \mathit{Traversable}\ t \Rightarrow (a \to \mathit{Int}) \to t\ a \to \mathit{Int} \\ \mathit{tsum}\,f = \mathit{unK} \circ \mathit{traverse}\ (K \circ f) \end{array}$$

5.4 Examples of traversal: shape and contents

As well as being parametrically polymorphic in the collection elements, the generic traversal introduced above is parametrized along two further dimensions: it is ad-hoc datatype-generic in the datatype being traversed, and parametrically datatype-generic in the idiom in which the traversal is interpreted. Specializing the latter to the lists-as-monoid idiom yields a generic *contents* operation, which is in turn the basis for many other generic operations, including non-monoidal ones such as indexing:

$$\begin{array}{l} \mathit{contents} :: \mathit{Traversable}\ t \Rightarrow t\ a \to [\,a\,] \\ \mathit{contents} = \mathit{unK} \circ \mathit{traverse}\ (K \circ \mathit{single}) \\ \mathit{single}\quad :: a \to [\,a\,] \\ \mathit{single}\ x = [\,x\,] \end{array}$$

This *contents* operation yields one half of Jay's decomposition of datatypes into shape and contents [72]. The other half is obtained simply by a map, which is to say, a traversal interpreted in the identity idiom:

$$\begin{array}{l} \mathit{shape} :: \mathit{Traversable}\ t \Rightarrow t\ a \to t\ () \\ \mathit{shape} = \mathit{unId} \circ \mathit{traverse}\ (\mathit{Id} \circ \mathit{bang}) \\ \mathit{bang} :: a \to () \\ \mathit{bang} = \mathit{const}\ () \end{array}$$

Of course, it is trivial to combine these two traversals to obtain both halves of the decomposition as a single function, but doing this by tupling in the obvious way entails two traversals over the data structure. Is it possible to fuse the two traversals into one? The product of idioms allows exactly this, yielding the decomposition of a data structure into shape and contents in a single pass:

$$\begin{array}{l} \mathit{decompose}' :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{Prod}\ \mathit{Id}\ (K\ [\,a\,])\ (t\ ()) \\ \mathit{decompose}' = \mathit{traverse}\ (\mathit{fork}\ (\mathit{Id} \circ \mathit{bang})\ (K \circ \mathit{single})) \end{array}$$

It is then a simple matter of removing the tags for the idiom product and the idioms themselves:

$$\begin{array}{l} \mathit{decompose} :: \mathit{Traversable}\ t \Rightarrow t\ a \to (t\ (), [\,a\,]) \\ \mathit{decompose} = \mathit{getPair} \circ \mathit{decompose}' \\ \mathit{getPair} :: \mathit{Prod}\ \mathit{Id}\ (K\ b)\ a \to (a, b) \\ \mathit{getPair}\ \mathit{xy} = (\mathit{unId}\ (\mathit{pfst}\ \mathit{xy}), \mathit{unK}\ (\mathit{psnd}\ \mathit{xy})) \end{array}$$

Moggi et al. [106] give a similar decomposition, but using a customized combination of monads; the above approach is arguably simpler.

A similar benefit can be found in the reassembly of a full data structure from separate shape and contents. This is a stateful operation, where the state consists of the contents to be inserted; but it is also a partial operation, because the number of elements provided may not agree with the number of positions in the shape. We therefore make use of both the *State* monad and the *Maybe* monad; but this time, we form their composition rather than their product. (As it happens, the composition of the *State* and *Maybe* monads in this way forms another monad, but that is not the case in general.)

The crux of the traversal is the partial stateful function that strips the first element off the list of contents, if this list is non-empty:

```
takeHead :: State [a] (Maybe a)
takeHead = do { xs ← get;
                case xs of
                  []       → return Nothing
                  (y : ys) → do { put ys; return (Just y) } }
```

This is a composite idiomatic value, using the composition of the two monadic idioms *State* [*a*] and *Maybe*; traversal in the composite idiom using this operation returns a stateful function for the whole data structure.

```
reassemble′ :: Traversable t ⇒ t () → State [a] (Maybe (t a))
reassemble′ = unComp ∘ traverse (λ() → Comp takeHead)
```

Now it is simply a matter of running this stateful function, and checking that the contents are entirely consumed.

```
reassemble :: Traversable t ⇒ (t (), [a]) → Maybe (t a)
reassemble (x, ys) = allGone (runState (reassemble′ x) ys)
allGone :: (Maybe (t a), [a]) → Maybe (t a)
allGone (mt, [])      = mt
allGone (mt, (_ : _)) = Nothing
```

5.5 Collection and dispersal

We have found it convenient to consider special cases of effectful traversals in which the mapping aspect is independent of the accumulation, and vice versa.

```
collect   :: (Traversable t, Idiom m) ⇒
               (a → m ()) → (a → b) → t a → m (t b)
collect f g = traverse (λa → pure (λ() → g a) ⊛ f a)
disperse :: (Traversable t, Idiom m) ⇒
               m b → (a → b → c) → t a → m (t c)
disperse mb g = traverse (λa → pure (g a) ⊛ mb)
```

The first of these traversals accumulates elements effectfully, but modifies those elements purely and independently of this accumulation. The C# iteration at the start of Section 5 is an example, using the idiom of the *State* monad to capture the counting:

$$\begin{aligned}&\mathit{loop} :: \mathit{Traversable}\ t \Rightarrow (a \to b) \to t\ a \to \mathit{State}\ \mathit{Int}\ (t\ b)\\ &\mathit{loop}\ \mathit{touch} = \mathit{collect}\ (\lambda a \to \mathbf{do}\ \{\, n \leftarrow \mathit{get}; \mathit{put}\ (n+1)\,\})\ \mathit{touch}\end{aligned}$$

The second kind of traversal modifies elements effectfully but dependent on the state, evolving the state independently of the elements. An example of this is a kind of converse of counting, labelling every element with its position in order of traversal.

$$\begin{aligned}&\mathit{label} :: \mathit{Traversable}\ t \Rightarrow t\ a \to \mathit{State}\ \mathit{Int}\ (t\ (a, \mathit{Int}))\\ &\mathit{label} = \mathit{disperse}\ \mathit{step}\ (,)\\ &\mathit{step} :: \mathit{State}\ \mathit{Int}\ \mathit{Int}\\ &\mathit{step} = \mathbf{do}\ \{\, n \leftarrow \mathit{get}; \mathit{put}\ (n+1); \mathit{return}\ n\,\}\end{aligned}$$

5.6 Backwards traversal

Unlike the case with pure maps, the order in which elements are visited in an effectful traversal is significant; in particular, iterating through the elements backwards is observably different from iterating forwards. We can capture this reversal quite elegantly as an *idiom adapter*, via the 'marker type' *Backwards*.

$$\begin{aligned}&\mathbf{newtype}\ \mathit{Backwards}\ m\ a = B\{\, \mathit{runB} :: m\ a\,\}\\ &\mathbf{instance}\ \mathit{Idiom}\ m \Rightarrow \mathit{Idiom}\ (\mathit{Backwards}\ m)\ \mathbf{where}\\ &\quad \mathit{pure}\ \ = B \circ \mathit{pure}\\ &\quad f \circledast x = B\ (\mathit{pure}\ (\lambda x\ f \to f\ x) \circledast \mathit{runB}\ x \circledast \mathit{runB}\ f)\end{aligned}$$

Informally, *Backwards m* is an idiom if *m* is, but any effects happen in reverse; in this way, the symmetric 'backwards' embedding of monads into idioms referred to in Section 5.2 can be expressed in terms of the forwards embedding given there. (Such marker types are generally a useful technique for selecting a particular ad-hoc datatype-generic implementation.)

An adapter can be parcelled up existentially:

$$\begin{aligned}&\mathbf{data}\ \mathit{IAdapter}\ m = \forall g.\ \mathit{Idiom}\ (g\ m) \Rightarrow\\ &\qquad\qquad\qquad \mathit{IAdapter}\ (\forall a.\ m\ a \to g\ m\ a)\ (\forall a.\ g\ m\ a \to m\ a)\\ &\mathit{backwards} :: \mathit{Idiom}\ m \Rightarrow \mathit{IAdapter}\ m\\ &\mathit{backwards} = \mathit{IAdapter}\ B\ \mathit{runB}\end{aligned}$$

and used in a parametrized traversal, for example to label backwards:

$$\begin{aligned}&\mathit{ptraverse} :: (\mathit{Idiom}\ m, \mathit{Traversable}\ t) \Rightarrow\\ &\qquad\qquad \mathit{IAdapter}\ m \to (a \to m\ b) \to t\ a \to m\ (t\ b)\\ &\mathit{ptraverse}\ (\mathit{IAdapter}\ \mathit{wrap}\ \mathit{unwrap})\ f = \mathit{unwrap} \circ \mathit{traverse}\ (\mathit{wrap} \circ f)\\ &\mathit{lebal} = \mathit{ptraverse}\ \mathit{backwards}\ (\lambda a \to \mathit{step})\end{aligned}$$

Of course, there is a trivial *forwards* adapter too, which can be used as a default.

```
newtype Forwards m a = F{ runF :: m a}
instance Idiom m ⇒ Idiom (Forwards m) where
  pure  = F ∘ pure
  f ⊛ x = F (runF f ⊛ runF x)
forwards :: Idiom m ⇒ IAdapter m
forwards = IAdapter F runF
```

5.7 Laws of traverse

The *traverse* operator is ad-hoc datatype-generic; one must define it independently for each *Traversable* datatype (although, as noted above, its definition is in principle derivable). The type class *Traversable* determines its signature, but in line with other instances of genericity by signature such as *Functor* and *Monad*, we should consider 'healthiness conditions' on the definition.

Free theorems The free theorem [117, 132] arising from the type of *dist* is

$$dist \circ fmap\ (fmap\ k) = fmap\ (fmap\ k) \circ dist$$

As corollaries, we get the following two free theorems of *traverse*:

$$\begin{aligned} traverse\ (g \circ h) &= traverse\ g \circ fmap\ h \\ traverse\ (fmap\ k \circ f) &= fmap\ (fmap\ k) \circ traverse\ f \end{aligned}$$

These laws are not constraints on the implementation of *dist* and *traverse*; they follow automatically from their types.

Composition We have seen that idioms compose: there is an identity idiom *Id* and, for any two idioms *m* and *n*, a composite idiom *Comp m n*. We impose on implementations of *dist* the constraint of respecting this compositional structure. Specifically, the distributor *dist* respects the identity idiom:

$$dist \circ fmap\ Id = Id$$

and the composition of idioms:

$$dist \circ fmap\ Comp = Comp \circ fmap\ dist \circ dist$$

As corollaries, we get analogous properties of *traverse*.

$$\begin{aligned} traverse\ (Id \circ f) &= Id \circ fmap\ f \\ traverse\ (Comp \circ fmap\ f \circ g) &= Comp \circ fmap\ (traverse\ f) \circ traverse\ g \end{aligned}$$

Both of these corollaries have interesting interpretations. The first says that *traverse* interpreted in the identity idiom is essentially just *fmap*, as mentioned in Section 5.3. The second provides a fusion rule for traversals, whereby two consecutive traversals can be fused into one. We use this fusion rule in Section 5.8.

Naturality We also impose the constraint that the distributor *dist* is *natural in the idiom*, as follows. An *idiom transformation* $\phi :: m\ a \rightarrow n\ a$ from idiom m to idiom n is a polymorphic function (natural transformation) that respects the idiom structure:

$$\begin{aligned} &\phi\ \mathit{pure}_m\ a &&= \mathit{pure}_n\ a \\ &\phi\ (\mathit{mf} \circledast_m \mathit{mx}) &&= \phi\ \mathit{mf} \circledast_n \phi\ \mathit{mx} \end{aligned}$$

(Here, the idiom operators are subscripted by the idiom, for clarity.) Then *dist* must satisfy the following naturality property: for idiom transformation ϕ,

$$\mathit{dist}_n \circ \mathit{fmap}\ \phi = \phi \circ \mathit{dist}_m$$

For example, $\mathit{pure}_m \circ \mathit{unId}$ is an idiom transformation from idiom *Id* to idiom m, because

$$\begin{aligned} &\quad \mathit{pure}_m \circ \mathit{unId} \circ \mathit{pure}_{\mathit{Id}} \\ &= \quad \{\ \mathit{pure}_{\mathit{Id}} = \mathit{Id}\ \} \\ &\quad \mathit{pure}_m \end{aligned}$$

and

$$\begin{aligned} &\quad \mathit{pure}_m\ (\mathit{unId}\ (\mathit{mf} \circledast_{\mathit{Id}} \mathit{mx})) \\ &= \quad \{\ \mathit{mf} \circledast_{\mathit{Id}} \mathit{mx} = \mathit{Id}\ ((\mathit{unId}\ \mathit{mf})\ (\mathit{unId}\ \mathit{mx}))\ \} \\ &\quad \mathit{pure}_m\ ((\mathit{unId}\ \mathit{mf})\ (\mathit{unId}\ \mathit{mx})) \\ &= \quad \{\ \text{pure homomorphism}\ \} \\ &\quad \mathit{pure}_m\ (\mathit{unId}\ \mathit{mf}) \circledast_m \mathit{pure}_m\ (\mathit{unId}\ \mathit{mx}) \end{aligned}$$

Therefore $\mathit{pure}_m \circ \mathit{unId}$ must commute with *dist*, in the following sense:

$$\mathit{dist}_m \circ \mathit{fmap}\ (\mathit{pure}_m \circ \mathit{unId}) = \mathit{pure}_m \circ \mathit{unId} \circ \mathit{dist}_{\mathit{Id}}$$

As a consequence, we get a 'purity law':

$$\mathit{traverse}\ \mathit{pure} = \mathit{pure}$$

because

$$\begin{aligned} &\quad \mathit{traverse}_m\ \mathit{pure}_m \\ &= \quad \{\ \mathit{traverse}\ \} \\ &\quad \mathit{dist}_m \circ \mathit{fmap}\ \mathit{pure}_m \\ &= \quad \{\ \mathit{unId} \circ \mathit{Id} = \mathit{id}\ \} \\ &\quad \mathit{dist}_m \circ \mathit{fmap}\ \mathit{pure}_m \circ \mathit{fmap}\ \mathit{unId} \circ \mathit{fmap}\ \mathit{Id} \\ &= \quad \{\ \text{assumption}\ \} \\ &\quad \mathit{pure}_m \circ \mathit{unId} \circ \mathit{dist}_{\mathit{Id}} \circ \mathit{fmap}\ \mathit{Id} \\ &= \quad \{\ \text{compositionality: } \mathit{dist}_{\mathit{Id}} \circ \mathit{fmap}\ \mathit{Id} = \mathit{Id}\ \} \\ &\quad \mathit{pure}_m \end{aligned}$$

This is an entirely reasonable property of traversal; one might say that it imposes a constraint of shape preservation. (But there is more to it than shape preservation: a traversal that twists pairs necessarily 'preserves shape', because pairs are Naperian, but still breaks this law.) For example, consider the following definition of *traverse* on binary trees, in which the two children are swapped on traversal:

$$\begin{array}{l}\textbf{instance}\ \mathit{Traversable}\ \mathit{Btree}\ \textbf{where}\\ \quad \mathit{traverse}\ f\ (\mathit{Tip}\ a) \quad\ = \mathit{pure}\ \mathit{Tip} \circledast f\ a\\ \quad \mathit{traverse}\ f\ (\mathit{Bin}\ t\ u) = \mathit{pure}\ \mathit{Bin} \circledast \mathit{traverse}\ f\ u \circledast \mathit{traverse}\ f\ t\end{array}$$

With this definition, $\mathit{traverse}\ \mathit{pure} = \mathit{pure} \circ \mathit{reflect}$, where *reflect* (defined in Section 2.7) reverses a tree, and so the purity law does not hold; this is because the corresponding definition of *dist* is not natural in the idiom. Similarly, a definition with two copies of $\mathit{traverse}\ f\ t$ and none of $\mathit{traverse}\ f\ u$ makes $\mathit{traverse}\ \mathit{pure}$ purely return a tree in which every right child has been overwritten with its left sibling. Both definitions are perfectly well-typed, but (according to our constraints) invalid.

On the other hand, the following definition, in which the traversals of the two children are swapped, but the *Bin* operator is flipped to compensate, is blameless.

$$\begin{array}{l}\textbf{instance}\ \mathit{Traversable}\ \mathit{Btree}\ \textbf{where}\\ \quad \mathit{traverse}\ f\ (\mathit{Tip}\ a) \quad\ = \mathit{pure}\ \mathit{Tip} \circledast f\ a\\ \quad \mathit{traverse}\ f\ (\mathit{Bin}\ t\ u) = \mathit{pure}\ (\mathit{flip}\ \mathit{Bin}) \circledast \mathit{traverse}\ f\ u \circledast \mathit{traverse}\ f\ t\end{array}$$

(Here, $\mathit{flip}\ f\ x\ y = f\ y\ x$.) The effect of the reversal is that elements of the tree are traversed 'from right to left', which we consider to be a reasonable, if rather odd, definition of *traverse*. The corresponding distributor is

$$\begin{array}{l}\mathit{distB}\ (\mathit{Tip}\ a) \quad\ = \mathit{pure}\ \mathit{Tip} \circledast a\\ \mathit{distB}\ (\mathit{Bin}\ t\ u) = \mathit{pure}\ (\mathit{flip}\ \mathit{Bin}) \circledast \mathit{distB}\ u \circledast \mathit{distB}\ t\end{array}$$

or equivalently

$$\begin{array}{l}\mathit{distB} = \mathit{foldB}\ f\ g\ \textbf{where}\\ \quad f\ \mathit{ma} \qquad = \mathit{pure}\ \mathit{Tip} \circledast \mathit{ma}\\ \quad g\ \mathit{mt}\ \mathit{mu} = \mathit{pure}\ (\mathit{flip}\ \mathit{Bin}) \circledast \mathit{mu} \circledast \mathit{mt}\end{array}$$

where *foldB* is the fold for *Btree* defined in Section 2.7, for which the free theorem turns out to be the following fusion law:

$$\begin{array}{l}\quad h \circ \mathit{foldB}\ f\ g = \mathit{foldB}\ f'\ g' \circ \mathit{fmap}\ h\\ \Leftarrow\\ \quad h\ (f\ a) = f'\ (h\ a) \wedge h\ (g\ a\ b) = g'\ (h\ a)\ (h\ b)\end{array}$$

We leave as a straightforward task for the reader the proof using this fusion law that

$$\mathit{distB} \circ \mathit{fmap}\ \phi = \phi \circ \mathit{distB}$$

Hence *distB* is natural in the idiom, and as a consequence the purity law applies to this right-to-left traversal.

Composition of monadic traversals Another consequence of naturality is a fusion law specific to monadic traversals. The natural form of composition for monadic computations is called *Kleisli composition*:

$$(\bullet) :: \mathit{Monad}\ m \Rightarrow (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c)$$
$$(f \bullet g)\ x = \mathbf{do}\ \{ y \leftarrow g\ x; z \leftarrow f\ y; \mathit{return}\ z \}$$

The monad m is *commutative* if, for all mx and my,

$$\begin{aligned} & \mathbf{do}\ \{ x \leftarrow mx; y \leftarrow my; \mathit{return}\ (x, y) \} \\ = {} & \mathbf{do}\ \{ y \leftarrow my; x \leftarrow mx; \mathit{return}\ (x, y) \} \end{aligned}$$

When interpreted in the idiom of a commutative monad m, traversals with $f :: b \to m\ c$ and $g :: a \to m\ b$ fuse:

$$\mathit{traverse}\ f \bullet \mathit{traverse}\ g = \mathit{traverse}\ (f \bullet g)$$

This follows from the fact that $\mu \circ \mathit{unComp}$ forms an idiom transformation from $\mathit{Comp}\ m\ m$ to m, for a commutative monad m with join operator μ. (The proof is straightforward, albeit a little messy.)

This fusion law for the Kleisli composition of monadic traversals shows the benefits of the more general idiomatic traversals quite nicely. Note that the corresponding more general fusion law for idioms in Section 5.7 allows two different idioms rather than just one; moreover, there are no side conditions concerning commutativity. The only advantage of the monadic law is that there is just one level of monad on both sides of the equation; in contrast, the idiomatic law has two levels of idiom, because there is no analogue of the μ operator of a monad for collapsing two levels to one (and besides, those two levels may be for different idioms).

We conjecture (but have not proved) that the monadic traversal fusion law also holds even if m is not commutative, provided that f and g themselves commute ($f \bullet g = g \bullet f$); but this no longer follows from naturality of the distributor in any simple way, and it imposes the alternative constraint that the three types a, b, c are equal.

No duplication Another constraint we impose upon a definition of *traverse* is that it should visit each element precisely once. For example, we consider this definition of *traverse* on lists to be bogus, because it visits each element twice.

```
instance Traversable [] where
  traverse f []       = pure []
  traverse f (x : xs) = pure (const (:)) ⊛ f x ⊛ f x ⊛ traverse f xs
```

Note that this definition satisfies the purity law above; but we would still like to rule it out.

This axiom is harder to formalize, and we do not yet have a nice theoretical treatment of it. One way of proceeding is in terms of indexing. We require that the function *labels* returns an initial segment of the natural numbers, where

$$\begin{aligned}
&\mathit{labels} :: \mathit{Traversable}\ t \Rightarrow t\ a \to [\mathit{Int}]\\
&\mathit{labels}\ t = \mathit{contents}\ \$\ \mathit{fmap}\ \mathit{snd}\ \$\ \mathit{fst}\ \$\ \mathit{runState}\ (\mathit{label}\ t)\ 0
\end{aligned}$$

Here, \$ denotes function application, and *label* is as defined in Section 5.5. The bogus definition of *traverse* on lists given above is betrayed by the fact that we get *labels* `"abc"` $= [1, 1, 3, 3, 5, 5]$, which is not an initial segment of the naturals.

5.8 Example

As a small example of fusion of traversals, we consider the familiar *repmin* problem [11]. The task here is to take a binary tree of integers, compute the minimum element, then replace every element of the tree by that minimum — but to do so in a single traversal rather than the obvious two. Our point here is not the circularity for which the problem was originally invented, but simply an illustration of the two kinds of traversal (mapping and accumulating) and their fusion.

Flushed with our success at capturing different kinds of traversal idiomatically, we might try computing the minimum in a monoidal idiom,

$$\begin{aligned}
&\mathbf{newtype}\ \mathit{Min}\ a = \mathit{Min}\{\mathit{unMin} :: a\}\\
&\mathbf{instance}\ (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Monoid}\ (\mathit{Min}\ a)\ \mathbf{where}\\
&\quad \emptyset \qquad = \mathit{Min}\ \mathit{maxBound}\\
&\quad x \oplus y = \mathit{Min}\ (\mathit{unMin}\ x\ \text{`}\mathit{min}\text{`}\ \mathit{unMin}\ y)\\
&\mathit{tmin}_1 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow a \to K\ (\mathit{Min}\ a)\ a\\
&\mathit{tmin}_1 = K \circ \mathit{Min}
\end{aligned}$$

and replacing in the idiom of the environment monad.

$$\begin{aligned}
&\mathit{trep}_1 :: a \to \mathit{Env}\ b\ b\\
&\mathit{trep}_1 = \lambda a \to \mathit{Env}\ \mathit{id}
\end{aligned}$$

These two compose elegantly (modulo the type isomorphisms):

$$\begin{aligned}
&\mathit{trepmin}_1 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Btree}\ a \to \mathit{Btree}\ a\\
&\mathit{trepmin}_1\ t = \mathit{unEnv}\ (\mathit{traverse}\ \mathit{trep}_1\ t)\ (\mathit{unMin}\ \$\ \mathit{unK}\ \$\ \mathit{traverse}\ \mathit{tmin}_1\ t)
\end{aligned}$$

However, the two traversals do not fuse: the first traversal computes the minimum and discards the tree, which then needs to be reintroduced for the second traversal.

Notice that $\mathit{trepmin}_1$ could be datatype-generic in the data structure traversed; the only constraint is that it should be *Traversable*.

$$\begin{aligned}
&\mathit{grepmin}_1 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a, \mathit{Traversable}\ t) \Rightarrow t\ a \to t\ a\\
&\mathit{grepmin}_1\ t = \mathit{unEnv}\ (\mathit{traverse}\ \mathit{trep}_1\ t)\ (\mathit{unMin}\ \$\ \mathit{unK}\ \$\ \mathit{traverse}\ \mathit{tmin}_1\ t)
\end{aligned}$$

The same observation applies to all versions of *trepmin* in this section; but to avoid carrying the *Traversable t* context around, we specialize to *Btree* throughout.

The problem with $\mathit{trepmin}_1$ is that the traversal that computes the minimum discards the tree. Apparently this first phase ought to retain and return the tree as well; this suggests using the idiom of the state monad. The state records the minimum element; the first traversal updates this state, and the second traversal reads from it.

$$\begin{aligned}
&\mathit{tmin}_2 :: \mathit{Ord}\ a \Rightarrow a \to \mathit{State}\ a\ a\\
&\mathit{tmin}_2\ a = \mathbf{do}\ \{\, b \leftarrow \mathit{get}; \mathit{put}\ (\mathit{min}\ a\ b); \mathit{return}\ a \,\}\\
&\mathit{trep}_2\ \ :: a \to \mathit{State}\ a\ a\\
&\mathit{trep}_2\ a\ \ = \mathit{get}
\end{aligned}$$

Again, traversals with tmin_2 and trep_2 compose.

$$\begin{aligned}
&\mathit{trepmin}_2 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Btree}\ a \to \mathit{Btree}\ a\\
&\mathit{trepmin}_2\ t = \mathit{fst}\ (\mathit{runState}\ \mathit{iteration}\ \mathit{maxBound})\\
&\quad \mathbf{where}\ \mathit{iteration} = (\mathit{traverse}\ \mathit{trep}_2 \bullet \mathit{traverse}\ \mathit{tmin}_2)\ t
\end{aligned}$$

But when we try to apply the fusion law for monadic traversals, we are forced to admit that the *State* monad is the epitome of a non-commutative monad, and in particular that the two stateful operations tmin_2 and trep_2 do not commute; therefore, the two traversals do not fuse.

There is a simple way to make the two stateful operations commute, and that is by giving them separate parts of the state on which to act. The following implementation uses a pair as the state; the first component is where the minimum element is accumulated, and the second component holds what is copied across the tree.

$$\begin{aligned}
&\mathit{tmin}_3 :: \mathit{Ord}\ a \Rightarrow a \to \mathit{State}\ (a, b)\ a\\
&\mathit{tmin}_3\ a = \mathbf{do}\ \{\, (a', b) \leftarrow \mathit{get}; \mathit{put}\ (\mathit{min}\ a\ a', b); \mathit{return}\ a \,\}\\
&\mathit{trep}_3\ \ :: a \to \mathit{State}\ (a, b)\ b\\
&\mathit{trep}_3\ a\ \ = \mathbf{do}\ \{\, (a', b) \leftarrow \mathit{get}; \mathit{return}\ b \,\}
\end{aligned}$$

Of course, the whole point of the exercise is that the two parts of the state *should* interact; but with lazy evaluation we can use the standard circular programming trick that originally motivated the repmin problem [11] to tie the two together, outside the traversal.

$$\begin{aligned}
&\mathit{trepmin}_3 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Btree}\ a \to \mathit{Btree}\ a\\
&\mathit{trepmin}_3\ t = \mathbf{let}\ (u, (m, \_)) = \mathit{runState}\ \mathit{iteration}\ (\mathit{maxBound}, m)\ \mathbf{in}\ u\\
&\quad \mathbf{where}\ \mathit{iteration} = (\mathit{traverse}\ \mathit{trep}_3 \bullet \mathit{traverse}\ \mathit{tmin}_3)\ t
\end{aligned}$$

Now, although the *State* monad is not commutative, the two stateful operations tmin_3 and trep_3 commute (because they do not interfere), and the two traversals may be fused into one.

$$\begin{aligned}
&\mathit{trepmin}'_3 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Btree}\ a \to \mathit{Btree}\ a\\
&\mathit{trepmin}'_3\ t = \mathbf{let}\ (u, (m, \_)) = \mathit{runState}\ \mathit{iteration}\ (\mathit{maxBound}, m)\ \mathbf{in}\ u\\
&\quad \mathbf{where}\ \mathit{iteration} = \mathit{traverse}\ (\mathit{trep}_3 \bullet \mathit{tmin}_3)\ t
\end{aligned}$$

Modifying the stateful operations in this way to keep them from interfering is not scalable, and it is not clear whether this trick is possible in general anyway. Fortunately, idioms provide a much simpler means of fusion. Using the same single-component stateful operations *tmin*$_2$ and *trep*$_2$ as above, but dispensing with Kleisli composition, we get the following composition of traversals.

```
trepmin₄ :: (Ord a, Bounded a) ⇒ Btree a → Btree a
trepmin₄ t = let (sf, m) = runState iteration maxBound
             in fst (runState sf m)
  where iteration = fmap (traverse trep₂) (traverse tmin₂ t)
```

Kleisli composition has the effect of flattening two levels into one; here we have to deal with both levels separately, hence the two occurrences of *runState*. The payback is that fusion of idiomatic traversals applies without side conditions!

```
trepmin′₄ :: (Ord a, Bounded a) ⇒ Btree a → Btree a
trepmin′₄ t = let (sf, m) = runState iteration maxBound
              in fst (runState sf m)
  where iteration = unComp $ traverse (Comp ∘ fmap trep₂ ∘ tmin₂) t
```

Note that the Kleisli composition of two monadic computations imposes the constraint that both computations are in the same monad; in our example above, both computing the minimum and distributing the result use the *State* monad. However, these two monadic computations are actually rather different in structure, and use different aspects of the *State* monad: the first writes, whereas the second reads. We could capture this observation directly by using two different monads, each tailored for its particular use.

```
tmin₅ :: (Ord a, Bounded a) ⇒ a → Writer (Min a) a
tmin₅ a = do {tell (Min a); return a}

trep₅ :: a → Reader a a
trep₅ a = ask
```

Here, *tell* adds a value to a monoidal state, returning unit, and *ask* retrieves the state.

```
newtype Writer s a = Writer{runWriter :: (a, s)}
tell :: Monoid s ⇒ s → Writer s ()

newtype Reader s a = Reader{runReader :: s → a}
ask :: Reader s s
```

The use of two different monads like this rules out Kleisli composition. However, idiomatic composition handles two different idioms (and hence two different monads) with aplomb.

```
trepmin₅ :: (Ord a, Bounded a) ⇒ Btree a → Btree a
trepmin₅ t = let (r, m) = runWriter iteration in runReader r (unMin m)
  where iteration = fmap (traverse trep₅) (traverse tmin₅ t)
```

These two traversals fuse in exactly the same way as before.

$$\begin{array}{l} \mathit{trepmin}'_5 :: (\mathit{Ord}\ a, \mathit{Bounded}\ a) \Rightarrow \mathit{Btree}\ a \rightarrow \mathit{Btree}\ a \\ \mathit{trepmin}'_5\ t = \mathbf{let}\ (r, m) = \mathit{runWriter}\ \mathit{iteration}\ \mathbf{in}\ \mathit{runReader}\ r\ (\mathit{unMin}\ m) \\ \quad \mathbf{where}\ \mathit{iteration} = \mathit{unComp}\ \$\ \mathit{traverse}\ (\mathit{Comp} \circ \mathit{fmap}\ \mathit{trep}_5 \circ \mathit{tmin}_5)\ t \end{array}$$

6 Conclusions

The material in these lecture notes owes much to the work of a number of colleagues. Section 2 builds on many discussions with colleagues in and around the *Datatype-Generic Programming* project at Oxford and Nottingham. My views on origami programming in Section 3 are based on ideas from the Algebra of Programming ('Squiggol') community, and especially the work of: Roland Backhouse and Grant Malcolm [92,7,6]; Richard Bird and Oege de Moor [9,10]; Maarten Fokkinga, Erik Meijer and Ross Paterson [32,101]; Johan Jeuring, Patrik Jansson, Ralf Hinze and Andres Löh [68,69,57,60,91]; and John Hughes [66]. The analogy between design patterns and higher-order datatype-generic programs discussed in Section 4 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 [39] and OOPSLA [40] later that year; the contribution of participants at those venues and at less formal presentations of the same ideas is gratefully acknowledged. The results reported in Section 5 are the outcome of joint work with Bruno Oliveira, and have benefited greatly from discussions with Conor McBride and Ross Paterson, whose work provided most of the technical results. To all of these people, and to the unnamed others who have also contributed, I am very grateful for encouragement, inspiration and insight. I would like to add final thanks to Andres Löh and Ralf Hinze for their extremely useful lhs2TEX translator.

References

1. Reference manual for the Ada programming language. American National Standards Institute, Inc., 1983. ANSI/MIL-STD-1815A-1983.
2. Martin Aigner and Günter M. Ziegler. *Proofs from The Book.* Springer-Verlag, third edition, 2004.
3. Andrei Alexandrescu. *Modern C++ Design.* Addison-Wesley, 2001.
4. Matt Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library.* Addison-Wesley, 1998.
5. 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.
6. Roland Backhouse and Paul Hoogendijk. Elements of a relational theory of datatypes. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, volume 755 of *Lecture Notes in Computer Science*, pages 7–42. Springer-Verlag, 1993.

7. Roland Carl Backhouse, Patrik Jansson, Johan Jeuring, and Lambert G. L. T. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115, 1998.
8. Michel Bidoit and Peter D. Mosses. CASL *User Manual*, volume 2900 of *Lecture Notes in Computer Science (IFIP Series)*. Springer, 2004.
9. Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.
10. Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.
11. Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.
12. Richard S. Bird. Lectures on constructive functional programming. In Manfred Broy, editor, *Constructive Methods in Computer Science*, pages 151–218. Springer-Verlag, 1988. Also available as Technical Monograph PRG-69, from the Programming Research Group, Oxford University.
13. Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Add generic types to the Java programming language. `http://www.jcp.org/en/jsr/detail?id=014`, April 2001. JSR 14.
14. Manuel Bronstein, William Burge, Timothy Daly, James Davenport, Michael Dewar, Martin Dunstan, Albrecht Fortenbacher, Patrizia Gianni, Johannes Grabmeier, Jocelyn Guidry, Richard Jenks, Larry Lambe, Michael Monagan, Scott Morrison, William Sit, Jonathan Steinbach, Robert Sutor, Barry Trager, Stephen Watt, Jim Wen, and Clifton Williamson. *Axiom: The Thirty-Year Horizon*. `http://wiki.axiom-developer.org/Mirrors?go=/public/book2.pdf`, 2003.
15. Peter Buchlovsky and Hayo Thielecke. A type-theoretic reconstruction of the Visitor pattern. *Electronic Notes in Theoretical Computer Science*, 155, 2005. 21st Conference on Mathematical Foundations of Programming Semantics.
16. Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
17. James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop*, pages 90–104, 2002.
18. Koen Claessen and John Hughes. Specification based testing with QuickCheck. In Gibbons and de Moor [45], pages 17–40.
19. Dave Clarke and Andres Löh. Generic Haskell, specifically. In Gibbons and Jeuring [47], pages 21–47.
20. CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science (IFIP Series)*. Springer, 2004.
21. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
22. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
23. Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Principles of Programming Languages*, pages 207–212, 1982.
24. Nancy A. Day, John Launchbury, and Jeff Lewis. Logical abstractions in haskell. In *Haskell Workshop*. Utrecht University Department of Computer Science, Technical Report UU-CS-1999-28, October 1999.

25. James C. Dehnert and Alexander 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.
26. Bruno César dos Santos Oliveira and Jeremy Gibbons. TypeCase: A design pattern for type-indexed functions. In Daan Leijen, editor, *Haskell Workshop*, 2005.
27. Jack Edmonds. Matroids and the Greedy Algorithm. *Mathematical Programming*, 1:125–136, 1971.
28. Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer, 1985.
29. Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer, 1990.
30. Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Department INF, Universiteit Twente, June 1994.
31. Maarten M. Fokkinga. Tupling and mutumorphisms. *The Squiggolist*, 1(4):81–82, June 1990.
32. Maarten M. Fokkinga and Erik Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.
33. Ira R. Forman and Scott Danforth. *Putting Metaclasses to Work.* Addison-Wesley, 1999. ISBN 0-201-43305-2.
34. Erich Gamma and Kent Beck. JUnit: Testing resources for extreme programming. `http://www.junit.org/`, 2000.
35. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
36. Neil Ghani, Tarmo Uustalu, and Varmo Vene. Build, augment and destroy, universally. In W.-N. Chin, editor, *Second Asian Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, 2004.
37. Jeremy Gibbons. Calculating functional programs. In Roland Backhouse, Roy Crole, and Jeremy 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.
38. Jeremy Gibbons. Origami programming. In Gibbons and de Moor [45], pages 41–60.
39. Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. `http://2005.ecoop.org/8.html`, June 2005. Tutorial presented at ECOOP.
40. Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. `http://www.oopsla.org/2005/ShowEvent.do?id=121`, October 2005. Tutorial presented at OOPSLA.
41. Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. In Ralf Hinze, editor, *Workshop on Generic Programming*, September 2006.
42. Jeremy Gibbons. Metamorphisms: Streaming representation-changers. *Science of Computer Programming*, 65:108–139, 2007.
43. Jeremy Gibbons, Roland Backhouse, Bruno Oliveira, and Fermín Reig. Datatype-generic programming project. `http://web.comlab.ox.ac.uk/oucl/research/pdt/ap/dgp/`, October 2003.
44. Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the Iterator pattern. In Tarmo Uustalu and Conor McBride, editors, *Mathematically-Structured Functional Programming*, July 2006.
45. Jeremy Gibbons and Oege de Moor, editors. *The Fun of Programming.* Cornerstones in Computing, ISBN 1-4039-0772-2. Palgrave, 2003.

46. Jeremy Gibbons, Graham Hutton, and Thorsten Altenkirch. When is a function a fold or an unfold? *Electronic Notes in Theoretical Computer Science*, 44(1), April 2001. Proceedings of Coalgebraic Methods in Computer Science.
47. Jeremy Gibbons and Johan Jeuring, editors. *Generic Programming.* Kluwer Academic Publishers, 2003.
48. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, September 1998.
49. Jeremy Gibbons, David Lester, and Richard Bird. Enumerating the rationals. *Journal of Functional Programming*, 16:281–291, 2006.
50. Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, 1993.
51. Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur.* PhD thesis, Université de Paris VII, 1972.
52. Sergei Gorlatch and Christian Lengauer. Parallelization of divide-and-conquer in the Bird-Meertens Formalism. *Formal Aspects of Computing*, 3, 1995.
53. Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *Object-Oriented Programming, Systems, Languages, and Applications*, October 2006.
54. Douglas Gregor and Sibylle Schupp. Making the usage of STL safe. In Gibbons and Jeuring [47], pages 127–140.
55. John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, Andreś Modet, and Jeannette M. Wing. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, New York, 1993.
56. Tatsuya Hagino. *A Categorical Programming Language.* PhD thesis, Department of Computer Science, University of Edinburgh, September 1987.
57. Ralf Hinze. Polytypic values possess polykinded types. In Roland Carl Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2000.
58. Ralf Hinze. Generics for the masses. In *International Conference on Functional Programming*, pages 236–243. ACM Press, 2004.
59. Ralf Hinze. Church numerals, twice! *Journal of Functional Programming*, 15(1):1–13, 2005.
60. Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Summer School on Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag, 2003.
61. Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. In this volume, 2006.
62. Ralf Hinze and Simon Peyton Jones. Derivable type classes. In *Haskell Workshop*, 2000.
63. Ralf Hinze and Andres Löh. Generic programming, now! In this volume, 2006.
64. C. A. R. Hoare. Notes on data structuring. In Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, APIC studies in data processing, pages 83–174. Academic Press, 1972.
65. Paul Hoogendijk and Roland Backhouse. When do datatypes commute? In Eugenio Moggi and Guiseppe Rosolini, editors, *Category Theory and Computer Sci-*

ence, volume 1290 of *Lecture Notes in Computer Science*, pages 242–260. Springer-Verlag, September 1997.
66. John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989.
67. Kenneth E. Iverson. *A Programming Language*. Wiley, 1962.
68. Patrick Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Principles of Programming Languages*, pages 470–482, 1997.
69. Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.
70. Patrik Jansson and Johan Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
71. Barry Jay and Paul Steckler. The functional imperative: Shape! In Chris Hankin, editor, *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 139–53, Lisbon, Portugal, 1998.
72. C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25(2-3):251–283, 1995.
73. Richard D. Jenks and Robert S. Sutor. *Axiom: The Scientific Computing System*. Springer-Verlag, 1992.
74. Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer Verlag, 1975.
75. Hal Jesperson. POSIX shell and utilities (p1003.2). `http://www.nic.funet.fi/pub/doc/posix/p1003.2/`, September 1991. Draft 11.2.
76. Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, 1995.
77. Mark P. Jones and Luc Duponcheel. Composing monads. Technical Report RR-1004, Department of Computer Science, Yale, December 1993.
78. Jevgeni Kabanov and Varmo Vene. Recursion schemes for dynamic programming. In Tarmo Uustalu, editor, *Mathematics of Program Construction*, volume 4014 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
79. Jennifer Kahn. It's alive! *Wired*, 10.03:72–77, March 2002.
80. Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation*, pages 1–12, Snowbird, Utah, 2001.
81. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
82. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *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.
83. David J. King and Philip Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Functional Programming, Glasgow 1992*. Springer, 1993.
84. Oleg Kiselyov and Ralf Lämmel. Haskell's Overlooked Object System. Technical Report cs/0509027, arXiv.org, September 2005.
85. Bernard Korte, László Lovász, and Rainer Schrader. *Greedoids*. Springer-Verlag, 1991.
86. Thomas Kühne. Internal iteration externalized. In Rachid Guerraoui, editor, *European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 329–350, 1999.

87. Ranko Lazic. *A Semantic Study of Data Independence with Applications to Model Checking.* D.Phil. thesis, Oxford University Computing Laboratory, 1999.
88. Ranko Lazic and David Nowak. On a semantic definition of data independence. In *Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, pages 226–240. Springer-Verlag, 2003. Technical Report CS-RR-392, Department of Computer Science, University of Warwick.
89. Barbara Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.
90. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
91. Andres Löh. *Exploring Generic Haskell.* PhD thesis, Utrecht University, 2004.
92. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
93. Ursula Martin and Tobias Nipkow. Automating Squiggol. In M. Broy and C. B. Jones, editors, *IFIP TC2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 233–246. North-Holland, 1990.
94. Conor McBride. Naperian functors. Personal communication by email, 5th April 2006.
95. Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, to appear.
96. James McKinna. Why dependent types matter. In *Principles of Programming Languages*, 2006.
97. John Meacham. DrIFT homepage. `http://repetae.net/~john/computer/haskell/DrIFT/`, 2004.
98. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
99. Lambert Meertens. Calculate polytypically! In H. Kuchen and S. D. Swierstra, editors, *Programming Language Implementation and Logic Programming*, volume 1140 of *Lecture Notes in Computer Science*, pages 1–16, 1996.
100. Lambert Meertens. Functor pulling. In Roland Backhouse and Tim Sheard, editors, *Workshop on Generic Programming*, Marstrand, Sweden, June 1998.
101. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.
102. Erik Meijer and Johan Jeuring. Merging monads and folds for functional programming. In Jeuring and Meijer [76].
103. Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
104. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML.* MIT Press, revised edition, 1997.
105. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.
106. Eugenio Moggi, Gianna Bellè, and C. Barry Jay. Monads, shapely functors and traversals. In M. Hoffman, D. Pavlovic, and P. Rosolini, editors, *Category Theory in Computer Science*, 1999.
107. David R. Musser and Alexander A. Stepanov. *The Ada Generic Library linear list processing packages.* Springer-Verlag New York, Inc., New York, NY, USA, 1989.

108. Peter Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6(1):1–17, January 1963.
109. Alberto Pardo. Fusion of recursive programs with computation effects. *Theoretical Computer Science*, 260:165–207, 2001.
110. Alberto Pardo. Combining datatypes and effects. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 171–209, 2005.
111. A. J. Perlis and K. Samelson. Preliminary report: International Algebraic Language. *Communications of the ACM*, 1(12):8–22, December 1958.
112. Simon Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.
113. Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for generalized algebraic data types. In *International Conference on Functional Programming*, 2006.
114. Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Principles of Programming Languages*, pages 71–84, 1993.
115. Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in Haskell. In *High Confidence Software and Systems Conference*. National Security Agency, April 2003.
116. John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.
117. John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier, 1983.
118. Fritz Ruehr. *Analytical and Structural Polymorphism Expressed Using Patterns over Types*. PhD thesis, University of Michigan, 1992.
119. Tim Sheard. Generic programming in Ωmega. In this volume, 2006.
120. Jeremy Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library.* Addison-Wesley, 2002.
121. Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *Programming Language Design and Implementation*, pages 73–84, 2005.
122. David B. Skillicorn. The Bird-Meertens Formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation*, volume 106 of *NATO ASI Series F*. Springer-Verlag, 1993.
123. STOP project. *International Summer School on Constructive Algorithmics, Hollum, Ameland*, 1989.
124. Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):1–49, 2000. Lecture notes from Summer School in Computer Programming, August 1967.
125. Walid Taha. A gentle introduction to multi-stage programming. In Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, number 3016 in Lecture Notes in Computer Science, pages 30–50. Springer-Verlag, 2004.
126. Erwin Unruh. Prime number computation. ANSI X3J16-94-0075/ISO WG21-462, 1994.
127. Tarmo Uustalu and Varmo Vene. Primitive (co)recursion and course-of-value (co)iteration. *Informatica*, 10(1):5–26, 1999.

128. A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68. *Acta Informatica*, 5(1-3), 1975.
129. Todd Veldhuizen. *Active Libraries and Universal Languages.* PhD thesis, Computer Science, Indiana University, 2004.
130. Varmo Vene and Tarmo 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.
131. Dimitrios Vytiniotis, Geoffrey Washburn, and Stephanie Weirich. An open and shut typecase. In *International Conference on Functional Programming*, 2004.
132. Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.
133. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
134. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.
135. Philip Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*, 1992. Also in [76].
136. Philip Wadler. How to solve the reuse problem? Functional programming. In *International Conference on Software Reuse*, pages 371–372. IEEE, 1998. `http://doi.ieeecomputersociety.org/10.1109/ICSR.1998.685772`.
137. Philip L. Wadler. The expression problem. Posting to java-genericity mailing list, 12th Nov 1998.
138. Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Communications of the ACM*, 9(6):413–432, June 1966.

7 Appendix: Java programs

Section 4.4 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 4.2 presents only the outline of a Java implementation of the same application. For completeness, this appendix presents the Java code.

7.1 Component

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

7.2 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);
  }
}
```

7.3 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);
  }
}
```

7.4 Iterator

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

7.5 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);
    }
  }
}
```

7.6 ParagraphIterator

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

7.7 Action

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

7.8 SpellCorrector

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

7.9 Visitor

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

7.10 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;
  }
}
```

7.11 Builder

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

7.12 InvalidBuilderId

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

7.13 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++;
        }
      }
    }
  }
}
```

7.14 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){
      if (records.isEmpty ()){
        records.addElement (new Record
          (nextId, nextId, s, ""));
        return nextId++;
      }
      else
        throw new InvalidBuilderId
          ("Duplicate root");
    } else {
      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;
        while (r.id ! = r.last){
          y = x;
          x = find (r.last, x);
          r = recordAt (x);
        }
        records.insertElementAt (new Record
          (nextId, nextId, s, indent + "    "), x + 1);
        recordAt (y).last = nextId;
        return nextId++;
      }
    }
  }
}
```

7.15 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){
      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 {
      PrintBuilder b = new PrintBuilder ();
      build (b);
      lines = b.getProduct ();
    }
    for (int i = 0; i < lines.length; i++)
      System.out.println (lines [i]);
  }
}
```