New Trends in Software Methodologies, Tools and Techniques
H. Fujita et al. (Eds.)
IOS Press, 2014

doi:10.3233/978-1-61499-434-3-513

From Computer Science to Software Engineering – A Programming-level Perspective

Paul BAILES[1], Leighton BROUGH, and Colin KEMP

School of Information Technology and Electrical Engineering
The University of Queensland
QLD 4072 AUSTRALIA

Abstract. An important step from computer science to a true discipline of software engineering is to impose discipline upon the behavior of practitioners. At the programming level, this is manifested by the constraint to use catamorphisms as the basic pattern of recursion/iteration. Catamorphisms exist for all regular recursive types and discipline (and simplify programming) to the selection of recursion pattern parameters. Reasoning about programs (for verification, transformation and optimization purposes) is similarly simplified. Catamorphisms are just one of a range of programming patterns explained by and understood in terms of mathematical category theory. The proposition that category theory provides the theory of programming (design as well as coding) is reinforced by the ability of modern programming languages to represent categorical abstractions in source code.

Keywords: Catamorphism, Category Theory, Fold, Functional, Recursion.

1. Introduction

One of the arguable distinctions between Science and Engineering is that the while the former may be governed in general terms by "Scientific Method", practice of the latter involves application of detailed procedures and techniques that are quite specific to the domain. In that context, it's arguable that genuine progress towards a discipline of Software Engineering (of which programming is only a component, but an essential one) should at least include restrictions on the range of programming tools and techniques otherwise available from Computer Science. This paper offers to contribute to that progress.

- Specifically: through the replacement of arbitrary recursive programming by a regime of patterns of recursion, in which the hierarchical relationship between patterns is exposed, including for the purpose of judicious extension by programmers;

[1] Corresponding author: paul@itee.uq.edu.au

- As a generalization of the above: through the use of category theory as a basis for the comprehension and application of further existing sophisticated programming patterns, as well as the creation of new such patterns.

While our specific result applies directly to the functional paradigm [1], the general result draws so much as well upon functional programming ("first class" functions, polymorphic types) that it's arguable that the functional paradigm is an inescapable part of the modern programming landscape, an argument that we see reflected in how the design of modern mainstream programming languages seem increasingly to embrace functional characteristics.

2. "Science" Versus "Engineering"

The difference between science and engineering underlies the development in programming that we aim for.

"Science" may be characterized by

- validation by disciplined experiment, of …
- artefacts in the form of theories developed however in relatively arbitrary style.

It may be observed that scientific practice is dominated by the conduct of many experiments to validate relatively few artefacts.

"Engineering" on the other hand may be characterized by

- production of numerous artefacts
- reliance for the validity/fitness-for-purpose of which on disciplined development using reliable methods and tools
- experimentation (i.e. testing) as a complement to the above.

The key difference between science and engineering thus emerges as the focus in engineering on disciplined development processes using reliable methods and tools (but developed of course using scientific results as appropriate). For example, an engineer may be called upon to design and equip a workshop capable of moving wheeled i.e. frictionless vehicles of at most 3500kg up an inclined ramp of 23 degrees (determined by architectural constraints). The very well-understood engineering abstraction of the inclined plane immediately informs our designer without further thought that

a) the ramp must support a force of 3500kg times gravitational constant 9.8 times cos 23 = 31,574N
b) the winch must be rated to 3500kg times 9.8 times sin 23 = 13,403N.

The underlying theory is in this case - Euclidean geometry and trigonometry, and Newton's theory of gravitation - is relatively straightforward (see Figure 1). The general point however is that the design engineer can simply invoke the appropriate design pattern by plugging in the parameters of the specific situation to obtain the desired result (in this the "inclined plane" pattern with parameters mass m and inclination θ) without having on every occasion to derive the result from first principles of the underlying theories or worse, by trial and error experimentation.

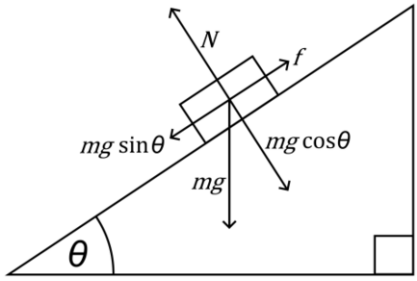

Figure 1. A civil & mechanical engineering design pattern - the inclined plane

While scientific laboratory practice is as highly-disciplined as, if not more so than any engineering laboratory/workshop, engineering is distinguished by the constraints it places on creative processes, relative to the uninhibited nature of scientific creativity.

Even though Computer Science is often not recognisably experimental, it remains "scientific" in that creation of artefacts is relatively undisciplined in its selection of processes, methods and tools. Transformation into Software Engineering entails the imposition of discipline in all of these categories. In this paper the explicit focus is on disciplined toolset that may be achieved at the programming language level, and thus implicitly on the methods and processes that complement this toolset.

3. Recursion

Recursion [3] is important not just theoretically but also as a practical basis for an engineering approach to programming. As the means by which significant, iterative operations on data structures are expressed in the applicative subset of functional programming, pure applicative style supports referential transparency [4] i.e. it enables derivation/validation and transformation/simplification/optimsation of programs formally by simple equational reasoning, rather than by ad hoc methods.

Consider the following recursive definitions of basic functions on lists (in the current de facto standard functional language Haskell [5]):

```
-- sum elements of a list
sum [] = 0                      -- sum.1
sum (x:xs) = x + sum xs         -- sum.2

-- append list ys at the end of list xs
[] ++ ys = ys                   -- ++.1
(x:xs) ++ ys = x:(xs ++ ys)     -- ++.2
```

Note:

- how the Haskell definitions use recursion-equation-style pattern-matching to effect top-level branching in functions;
- the facility to define operators (e.g. ++) as well as named function in this way
- .the ':' operator prepends element x to the head of list xs (typically read/pronounced "cons")
- comments identify each defining equation for subsequent reference below.

The property e.g. sum (xs ++ ys) = sum xs + sum ys can conceivably be used to simplify a program where the sum of a combined list xs ++ ys can be obtained from the previously-calculated sums of individual lists xs and ys. This property can be relied upon by programmer because it can be proved mathematically, by induction on xs as follows.

- Base case on xs = [ ]:

```
sum ([] ++ ys)
= (++.1)
sum ys
= (0 is identity for +)
0 + sum ys
= (sum.1)
sum [] + sum ys
```

- Induction on (x:xs):

```
sum ((x:xs) ++ ys)
= (++.2)
sum (x:(xs ++ ys))
= (sum.2)
x + sum(xs ++ ys)
= (inductive hypothesis)
x + sum xs + sum ys
= (sum.2)
sum (x:xs) + sum ys
```

Bird and Wadler [4] for instance give many more examples.

4. Recursion Patterns

Even though recursive definitions are compatible with the equational reasoning that is one of the hallmarks of an engineering approach to programming, the complexity entailed in developing and comprehending recursive definitions from first principles is less than ideal. Subprograms (subroutines, procedures, functions, macros, etc.) and design patterns more generally discipline and simplify programming by encapsulating various kinds of patterns of computation which can then be instantiated with the parameters of specific application circumstances. Accordingly, it's desirable (pursuing the "Engineering" ideal above) to identify and use a class of subprogram - recursion patterns, specifically to encapsulate patterns of recursion.

4.1. "Foldr" Pattern

Recursive functions are derived from recursion patterns by instantiating the patterns with operands germane to the functions. A popular (and as we shall see fundamental) recursion pattern, seen here in the context of lists, is

```
foldr op b [] = b
foldr op b (x:xs) = op x (foldr xs op b)
```

Thus we can redefine sum, ++ by simply by supplying the appropriate operands to foldr:

```
sum xs = foldr (+) 0 xs

xs ++ ys = foldr (:) ys xs
```

We thus have the basis of a new, disciplined and thus "engineering" style of programming. Instead of fabricating an explicit recursion, we simply "fill in the blanks" signified by the arguments to foldr. For example:

```
length xs = foldr (\_ lxs -> lxs+1) 0

reverse xs = foldr (\x rxs -> rxs ++ [x]) []

sort xs = foldr insert [] xs
-- insert x sxs = ... defined further below
-- inserts x into position in a sorted list
```

The residual creativity required for defining a function in this style simply entails two items:

1. the binary operation op that combines the head of a list with the result of applying the function to the tail of the list;
2. the base value b of the function for the empty list.

For example, in the syntheses of the above, all that is required is to implement the specifications for op and b in each case, e.g.:

- to append (++) a list ys at the end of a list xs:
  - op combines the list head with the result of appending ys to the list tail, i.e. the (:) operator
  - b is the result of appending ys to [], i.e. ys
- to reverse a list xs:
  - op combines the result rxs of reversing the tail of xs with the list head x, i.e. rxs ++ [x]
  - b is the result of reversing [], i.e. []
- to sort a list xs:
  - op inserts the list head into position in the sorted list tail
  - b is the result of sorting [], i.e. []
- etc. etc. etc.

4.2. Reasoning about "foldr"

As well as simplifying programming, recursion patterns such as foldr can be associated with theorems (such as the universal property, and especially fusion [6]) that simplify equational reasoning. Fusion obviates the need for explicit induction in proofs (just as foldr obviates the need for explicit recursion in definitions) and avoids potential difficulties, e.g. identification of the appropriate variable as basis for induction.

The fusion theorem is an implication:

h w = v ∧ h (g a b) = f a (h b)

→

h . foldr g w = foldr f v

By comparison with the above induction, e.g. for the proof of

```
sum (xs ++ ys) = sum xs + sum ys
```

to use fusion we first rewrite the proof requirements so that fusion is more clearly applicable:

```
foldr (+) 0 . foldr (:) ys = (+) (sum ys) . foldr (+) 0
```

Next, observe that fusion is applicable

1. to the LHS of this putative identity, where:

```
h = foldr (+) 0
g = (:)
w = ys
```

It follows from the fusion antecedents and definition of foldr that

```
v = sum ys
f = (+).
```

2. to the RHS of this putative identity, where:

```
h = (+) sum ys
g = (+)
w = 0
```

It follows from the fusion antecedents, definition of foldr and the identities

```
sum ys + 0 = sum ys
sum ys + (a+b) = a + (sum ys +b)
```

that

```
v = sum ys
f = (+)
```

The evident identity of the antecedents (f and v) of fusion on both sides of the putative identity completes the proof.

The foregoing is designed to exhibit a fusion-based proof as might be presented for experienced readers for comparison with induction. For a more detailed step-by-step explanation of fusion in tutorial style, see "Appendix - Fusion in Detail" below.

4.3. Catamorphisms - Generalising "Foldr" beyond Lists

Foldr is an example, specialized to lists, of a general pattern of recursion on regular recursive types - the catamorphism [7].

Consider the types

```
data Nat = Succ Nat | Zero

data BinT a = Brn (BinT a) (BinT a) | Tip a | Emt
```

Their catamorphic recursion patterns are respectively

```
catN s z (Succ n) = s (catN s z n)
catN s z Zero = z

catBt b t e (Brn lt rt) = b (catBt b t e lt) (catBt b t e rt)
```

```
catBt b t e (Tip x) = t x
catBt b t e Emt = e
```

Informally, in object-oriented terms, the operands of a catamorphism are servers, one for each distinct constructor or shape of data in the type. The catamorphism moves recursively through the instance to which it has been applied, applying the appropriate server for each constructor occurrence it encounters, applying recursively to recursive structure components, e.g. see in the above how catBt is applied to the BinT's that are the operands of Brn. (See also "Category Theory as Fundamental" below for a formal characterization of catamorphisms in general.)

As with lists, recursive functions on these types are engineered simply by supplying the appropriate catamorphic operands:

- the residual creativity required for defining a function on BinT simply entails three items:
  - the binary server b that combines the results of applying the function to the subtrees;
  - the unary server t that handles the value in the tip of a tree;
  - the base value e of the function for the empty tree.
- the residual creativity required for defining a function on Nat simply entails two items:
  - the unary server s that handles the results of applying the function to the predecessor value;
  - the base value z of the function for Zero.

For example

```
-- flatten bt into a list of Tip values
flatten bt = catBt (++) (\x->[x]) [] bt

-- count the number of Tips
tipcount bt = catBt (+) (\_->1) 0 bt

-- define basic arithmetic operations
add a b = catN Succ b a
mul a b = catN (add b) Zero b

-- etc.
```

In all these cases (including list catamorphism foldr), the programmer is relieved of the responsibility (and discretion) of managing the pattern of recursion, instead having (and being allowed) only to focus on the specifics of the function in question.

The definition of a fusion property for catN and catBt that simplifies inductive proofs is left as an exercise for the committed reader.

4.4. Other Recursion Patterns

There are situations when it's convenient to think of recursion in terms of other patterns besides catamorphism/foldr. Nevertheless as with catamorphisms, recursive functions are engineered simply by supplying the appropriate operands to these recursion patterns. Among many others, for example in the list context:

- list reversal may be accomplished with more apparent efficiency, by comparison with reverse above, in terms of foldl (where combination of

elements with the op argument is from the left rather than from the right with foldr)

```
revl xs = foldl (\rxs x -> x:rxs) [] xs

where

foldl op sofar [] = sofar
foldl op sofar (x:xs) = foldl op (op sofar x) xs
```

In the synthesis of revl, all that is required is to implement the specifications for op and sofar:

- op combines the result of reversing the list so far, with the current element, i.e. with the : operator.
- the top-level value of sofar is the result of reversing [], i.e. []

- inserting an element into sorted position a list may be accomplished in terms of list paramorphism paraL (where the unprocessed list tail is also available to the op operand, instead of just the recursively processed list tail as with list catamorphism/foldr)

```
insert e =
  paraL
  (\x xs exs -> if e<x then e:x:xs else x:exs)
  [e]

where

paraL op b []= b
paraL op b (x:xs) = op x xs (paraL op b xs)
```

In the synthesis of insert, all that is required is to implement the specifications for op and b:

- op combines the head of the list x and the unprocessed list tail xs with the result exs of inserting e into the list tail xs, so accordingly makes a choice based on the ordering between e and x;
- b is the result of inserting e into position in [], i.e. [e].

Where meaningful, counterparts to these patterns exist for other types, e.g.

- paramorphism on binary trees

```
paraBt b t e (Brn lt rt) =
       b lt (paraBt b t e lt) rt paraBt b t e rt)
paraBt b t e (Tip x) = t x
paraBt b t e Emt = e
```

- paramorphism on natural numbers Nat above can be used to define factorials:

```
paraN s z (Succ n) = s n (paraN s z n)
paraN s z Zero = z

where

fact n = paraN (\n n'-> mul (Succ n) n') (Succ Zero) n
```

In the synthesis of fact, all that is required is to implement the specifications for s and z:

- op combines the natural number (Succ n) with the factorial of its predecessor n, i.e. with the multiplication;
- z is the factorial of Zero, i.e. 1 (rendered as "Succ Zero").

Again, the definition of analogues to the fusion property that simplify inductive proofs for these patterns are left as exercises for the committed reader!

5. Catamorphic Patterns as Sufficient Basis

A key pragmatic consequence however of our basic premise (that recursion patterns are preferable, to the exclusion of "raw" recursion, as a basis for programming), is that the definition of new recursion patterns, and not just the definition of programs/functions, must be able to be based on existing, more basic patterns, i.e. catamorphisms. This allows us to avoid explicit recursion entirely, except for the basic definition of the catamorphism pattern.

5.1. Simulating Recursion Patterns

Catamorphisms suffice to define at least any function provably-terminating in second-order arithmetic [8], i.e. practically-speaking any reasonable function other than a Universal Turing Machine or other programming language interpreter. Hence they are theoretically-valid as a basis for recursion-pattern-based programming and software engineering.

Pragmatically, the functions and algorithms defined by these other recursion patterns are plausibly definable as catamorphisms, just as are basic functions such as sum, ++, length, flatten, add, mul, etc. That is, the patterns of computation embodied in foldl, paraL etc. can be simulated by foldr, where the catamorphic result is enhanced in a rational manner in order to simulate the other pattern.

For example:

- simulating foldl in the context of rev

```
revl xs =
    foldr
    (\x xs' -> (\sofar -> xs' (x:sofar)))
    (\sofar -> sofar)
    xs
    []
```

  The foldr result is enhanced to a function xs' parameterized by a context i.e. of reverse of the list so far, the initial value being set by the application of the top level result to [] and subsequent values maintained by applying lower-level results with the current head element prepended;

- simulating paraL in the context of insert

```
insert e xs =
   fst $ foldr
   (\x (exs, xs) ->
      (if e<x then e:x:xs else x:exs,x:xs)
   )
   ([e],[])
   xs
```

The foldr result at each level is enhanced to a pair comprised by (list with e inserted, list without e inserted) the ultimate outcome being obtained by selecting the first element of the top-level result. The difficulty is thereby overcome of giving the op operand to foldr access to the unprocessed tail of the current list.

See [9] for further explanations of these definitions and their rationales.

5.2. Synthesis of Recursion Patterns

The foregoing shows how to simulate other recursion patterns by catamorphism (foldr) in specific contexts. The general case, i.e. of the recursion patterns themselves, may be achieved by abstracting from the catamorphism-based counterparts of the specific functions. Specifically we abstract from the catamorphism operands, thus securing catamorphisms in their role as the basis of other recursion patterns.

For example:

- simulating foldl by foldr in general

```
foldl op b xs =
   foldr(\x xs' -> (\c -> xs' (op c x))) (\c -> c) xs b
```

by comparison with the specifics of revl we replace
  - the notion of the reversed list so far by a generic context parameter c
  - the combining operator (:) by the foldl pattern's generic op
  - the initial context value by the foldl pattern's generic b

- simulating paraL by foldr in general

```
paraL op b xs =
   fst $ foldr (\x (rxs,xs)->(op x xs rxs, x:xs))(b,[]) xs
```

by comparison with the specifics of insert we replace
  - the notion of the list with e inserted by a generic result rxs
  - the choice of the list with e correctly inserted by the foldl pattern's generic op
  - the initial result value by the foldl pattern's generic b.

The common development from the specifics (e.g. respectively revl, insert) is that the generic pattern operands op, b now appear in the operand expressions to foldr. The original definitions of revl and insert in terms of foldl and paraL patterns respectively are equally valid using the above foldr-based definitions of the patterns.

Similarly we can synthesise generic definitions for recursion patterns on other types in terms of their catamorphisms, e.g.

```
paraBt b t e bt =
   fst $ cataBt
   (\(rlt, lt), (rrt, rt) -> (b lt rlt rt rrt, Brn lt rt))
   (\a -> (t a, Tip a))
   (e, Emt)
   bt

paraN s z n =
   fst $ cataN (\(rn, n) -> (s n rn, Succ n)) (z, Zero) n
```

Observe how in each case, as with list paramorphisms, the catamorphism is enhanced:

- to yield a pair (main result at this level, original structure at this level)

- to accept these pairs as input wherever the lower-level catamorphism result is required.

6. Generalisation beyond Catamorphisms

Catamorphisms and other derived recursion patterns represent just one facet of a wide class of programming patterns that can be represented in source code.

6.1. Category Theory is Fundamental

The conceptual integrity of how catamorphisms (and anamorphisms - see Corecursion and Corecursion Patterns below) are applicable to all regular recursive datatypes is grounded in terms of the branch of abstract algebra known as Category Theory [10], which can be thought of as the science of pure structure, independent of the details of specific structures. As a result, category theory is well-suited to characterising the generic structures that abound in modern software engineering in particular generic/polymorphic types and functions. Thus, the catamorphic recursion pattern for any type is formally the least (i.e. most general) homomorphism from the pattern algebra of the type (i.e. that defines the shape of the data in the type) to any other algebra (i.e. the polymorphic target type of the recursion pattern). From this abstract characterization [11] can be derived practical results including the specification of the catamorphic pattern for each datatype and the transformational properties (e.g. fusion) by which programs can be verified or optimized [12].

Significantly, using the algebraic abstractions supported by the N-rank polymorphism extensions to the Haskell type system, a generic catamorphism operator reflecting this categorical characterisation can actually be expressed in Haskell source code, as per Uustalu et al [13]. First, they provide some general algebraic abstractions:

```
type Algebra f a = f a -> a
class Functor f where -- from Haskell prelude
   fmap :: (a -> b) -> f a -> f b
```

Notably including a fixpoint operator for data types:

```
newtype Mu f = InF { outF :: f (Mu f) }
```

The above algebraic definition of a generic catamorphism as homomorphism is thus directly encoded as follows.

```
cata :: Functor f => Algebra f a -> Mu f -> a
cata f = f . fmap (cata f) . outF
```

where ‘f’ represents the target algebra of the catamorphism.

Thus, e.g. list catamorphisms (foldr) as above can be re-written as follows. First, the pattern functor type of the polymorphic list type:

```
data Lf a lf = Nl | (Cns a lf)
instance Functor (Lf a) where
   fmap g Nl = Nl
   fmap g (Cns x xs) = Cns x (g xs)
```

Next, the actual recursive List type is the least fixed point of its pattern functor type:

```
type List a = Mu (Lf a)
```

Its constructors correspondingly have to be mapped from the pattern functor constructors into the recursive list type:

```
nil = Inf Nl
cons x xs = Inf (Cns x xs)
```

Finally, the definition of the counterpart to foldr, the catamorphism pattern on type List, simply applies the generic catamorphism cata to a generic target algebra for List:

```
catL o b = cata phi where
   phi Nl = b
   phi (Cns x xs) = o x xs
```

6.2. Corecursion and Corecursion Patterns

Just as recursion processes a recursive data structure (such as a natural number, list, tree, etc.), so does corecursion [14] by contrast build a similar such structure.

The basic corecursion pattern is the anamorphism, for lists often called "unfold" and definable as

```
unfold next ended seed =
   if ended seed then []
   else
      let (nextelt, nextseed) = next seed
      in nextelt : unfold next ended nextseed
```

For example, the Fibonacci sequence is definable as

```
unfold (\(fa,fb)->(fa,(fb,fa+fb))) (\_->False) (0,1)
```

Formally, anamorphisms are simply the categorical duals of catamorphisms and with equally-significant (if perhaps as-yet unappreciated) broad applicability [15]; space precludes further treatment here.

6.3. Categorical Abstractions in General

Another variety of category-theoretic abstraction is that of the programming patterns provided in terms of typeclasses e.g.: Functor as used in the definition of generic catamorphisms above; the famous Monad (model for side-effectful input-output in Haskell); and many others [16]. The abstractions supported by these patterns comprise an interface of operations (as in abstract data types, which can be enforced by Haskell) together with laws (currently beyond Haskell's type-enforcement capabilities) that, if maintained, can be used to predict the behavior of these abstractions.

What all these category-theoretic abstractions (catamorphisms, anamorphisms, functors, monads, other typeclass-based abstractions etc.) have in common is that each offers a characterization of highly-abstract computation, e.g.: catamorphisms capture a general pattern of recursion independent of the specific nature of any regular recursive type; monads capture a pattern of passing side-effects between successive stages of a computation. By "highly abstract" we mean where the specific datatypes and

operations used in a computation are generic i.e. unknown to the programmer of the abstraction.

Category theory can be thought of as the science of abstract structures, and as exemplified by the above supplies a rich source of concepts for the programmer: libraries of existing category-theoretic abstractions; but in addition a new mindset of programming that facilitates not just the use of these sophisticated abstractions but also the conception of new ones as actual source code artefacts.

7. Conclusions

7.1. Primary Conclusion

Primarily, we have indicated how a fundamental aspect of programming - processing recursive data structures - can be significantly simplified by adopting a disciplined engineering approach using established design patterns. Instead of general recursion, we instead rely on recursion patterns based on catamorphisms and the mere provision of operands to them. This simplification is evocative of earlier efforts to impose discipline on programming through restrictions on general control structures [17] and list structure processing [18]. We contend however that the discipline we advocate is more far-reaching. Restrictions on general control structures just do not give much guidance about how to process data. Restrictions on how lists are processed, in essence to catamorphisms and a few of the other catamorphism-expressible patterns, just don't address other data structures. Our approach embraces general catamorphisms as the basic class of recursion pattern which are ultimately applicable to all regular recursive types and with expressiveness that is practically universal. A key example of this universality is how exploiting higher-order operands to catamorphisms allows the synthesis of the other pragmatically-convenient recursion patterns.

7.2. Secondary Conclusions

At a deeper level however, the theory of catamorphisms that supports the above is grounded in the broader or meta-theoretical framework of category theory, in which programming patterns for applications beyond data structure processing can be conceived. The expressive power of modern (functional) programming languages means that these patterns can be defined as source-code-level abstractions and instantiated by programmers for different contexts of functions and data.

The implications are two-fold: one for language designers, and one for programmers/software engineers.

7.2.1. Mainstream Programming Languages need to embrace Functional Programming

The first of these validates the evident direction of modern programming language design, notably the increasing mainstreaming of functional capabilities into practical programming languages (e.g. Java8 [19], Scala [20]), which means that category-theory-based programming can be expected to be accessible to the mainstream software engineering community before long if not already. Further, these powerful abstraction mechanisms significantly increase the range of software engineering concepts

including even design patterns that can be expressed in source code [21], and thus with more assured integrity. In other words, an increasing amount of software engineering design can be conducted within and expressed in terms of programming languages and supported by validation technology such as type-checking. This if anything makes programming an even more critical part of the overall software engineering landscape.

7.2.2. Software Engineers need to understand Category Theory

The second and maybe even deeper implication is that category theory now needs to become an essential part of software engineering education. Granted, the various category-based abstractions covered above (catamorphisms, and the rest) can indeed be approached without explicit knowledge of the underlying theory. Indeed, extraordinary programmers may hitherto have experienced and applied their own intuitions corresponding to category-theoretic understandings of programming [22], and some of these abstractions were originally developed without reference to theory (foldr is at least as old as the "reduce" operator of APL [23]). However, in order to use these abstractions most effectively a deeper understanding is important in order to know their boundaries, i.e. the limits on their applicability and how they combine, or even to create new abstractions. Returning to our inclined plane design pattern analogy, the need for the civil engineer to understand that friction invalidates the pattern is reflected in a software engineer's need to know that the monadic "join" operator needs to be associative.

Perhaps decisively, the realisation of categorical abstractions as concrete source code artefacts in contemporary programming languages as demonstrated above, makes the empowerment of ordinary programmers with categorical insights an idea whose time has come.

8. Appendix - Fusion in Detail

Recall

1. the foldr-based definition of sum

```
sum xs = foldr (+) 0 xs
```

2. the foldr-based definition of ++

```
xs ++ ys = foldr (:) ys xs
```

3. and that fusion is an implication:

$h\ w = v \wedge h\ (g\ a\ b) = f\ a\ (h\ b) \rightarrow h\ .\ foldr\ g\ w = foldr\ f\ v$

So, to prove

```
sum (xs ++ ys) = sum xs + sum ys
```

first rewrite both sides of the putative equality in a form amenable to fusion.

- LHS:

```
sum (xs ++ ys)
= foldr (+) 0 (foldr (:) ys xs)
= (foldr (+) 0 . foldr (:) ys) xs
```

- RHS

```
sum xs + sum ys
= sum ys + sum xs
= (+) (sum ys) (sum xs)
= ((+) (sum ys) . sum) xs
```

Thus together:

```
(foldr (+) 0 . foldr (:) ys) xs = ((+) (sum ys) . sum) xs
```

Now because f x = g x iff f = g, we simplify ("eta conversion" in functional programming terminology) giving the proof obligation:

```
foldr (+) 0 . foldr (:) ys = (+) (sum ys) . sum
```

8.1. Transformation of LHS

Thus, observe that the LHS of the proof obligation

```
foldr (+) 0 . foldr (:) ys
```

corresponds to the LHS of the consequent of the fusion theorem, i.e. where:

```
h = foldr (+) 0
g = (:)
w = ys
```

In order to satisfy each of the conjuncts of the fusion antecedent, we calculate v and f respectively:

```
h w = foldr (+) 0 ys = sum ys = v
```

and (in which a, b are free variables):

```
h (g a b)
= foldr (+) 0 ((:) a b)
= (+) a (foldr (+) 0 b)
= f a (h b)
```

i.e.

```
f = (+)
```

Hence the LHS of our proof obligation can be rewritten into the form of the RHS of the fusion consequent:

```
foldr (+) (sum ys)
```

8.2. Transformation of RHS

Similarly, observe that expansion of the second occurrence of sum on the RHS of the proof obligation

```
(+) (sum ys) . foldr (+) 0
```

also corresponds to the LHS of the consequent of the fusion theorem, in this case where:

```
h = (+) (sum ys)
```

```
g = (+)
w = 0
```

Again, in order to satisfy each of the conjuncts of the fusion antecedent, we calculate v and f respectively:

```
h w = (+) (sum ys) 0 = sum ys = v
```

and

```
h (g a b)
= (+) (sum ys) ((+) a b)
= sum ys + a + b
= a + (sum ys + b)
= (+) a ((+) (sum ys) b)
= f a (h b)
```

i.e.

```
f = (+)
```

Hence the RHS of our proof obligation can be rewritten into the form of the RHS of the fusion consequent:

```
foldr (+) (sum ys)
```

8.3. Conclusion

Subsitution of the results of applying fusion to both sides of the proof obligation leads to an evident identity:

```
foldr (+) (sum ys) = foldr (+) (sum ys)
```

hence completing the proof.

References

[1] Hughes, J., Why Functional Programming Matters, *The Computer Journal*, vol. 32, no. 2, pp. 98-107 (1989).
[2] Sommerville, I., *Software engineering* (9th ed.). Pearson (2011).
[3] Kleene, S.C., *Introduction to Metamathematics*, Van Nostrand Rheinhold (1952).
[4] Bird, R. and Wadler, P., *An Introduction to Functional Programming*, Prentice-Hall (1988).
[5] *The Haskell Programming Language, http://www.haskell.org/haskellwiki/Haskell*, accessed 24 January 2014.
[6] Hutton, G., A Tutorial on the Universality and Expressiveness of Fold, *Journal of Functional Programming*, vol. 9, pp. 355-372 (1999).
[7] Meijer, E., Fokkinga, M., and Paterson, R., Functional Programming with Bananas, Lenses, Envelopes, and Barbed Wire, *Proc. FPCA 1991*, pp. 124-144 (1991).
[8] Reynolds, J., Three approaches to type structure, *Mathematical Foundations of Software Development, LNCS*, vol. 185, pp. 97-138 (1985).
[9] Bailes, P. and Brough, L., Making Sense of Recursion Patterns, *Proc. 1st FormSERA: Rigorous and Agile Approaches*, IEEE, pp. 16-22 (2012).
[10] Lawvere, F.W. and Schanuel, S., *Conceptual Mathematics. A first introduction to categories* (2nd ed.), Cambridge (2009).
[11] Backhouse, R., Jansson, P., Jeuring, J. and Meertens, L.: Generic Programming - An Introduction. In Swierstra, S., Henriques, P. and Oliveira, J. (eds.), *Advanced Functional Programming*, LNCS, vol. 1608, pp. 28-115 (1999)

[12] Launchbury, J. and Sheard, T., Warm Fusion: Deriving Build-Catas from Recursive Definitions, *Proc. FPCA 1995*, pp. 314-323, ACM (1995).
[13] Uustalu, T., Vene, V., Pardo, A., Recursion schemes from Comonads, *Nordic Journal of Computing*, Volume 8 , Issue 3, pp. 366-390 (2001).
[14] Bird, R. Using circular programs to eliminate multiple traversals of data, *Acta Informatica* 21 (3): 239-250 (1984).
[15] Rutten, J., Universal coalgebra: a theory of systems, *Theoretical Computer Science* 249, pp. 3-80, (2000).
[16] Yorgey, B., The Typeclassopedia, *Monad.Reader http://themonadreader.wordpress.com*/, 13, 1989
[17] Dijkstra, E., Goto Statement Considered Harmful, *Comm. ACM*, vol. 11, pp. 147-148 (1968).
[18] Backus, J., Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs, *Comm. ACM*, vol. 9 (1978).
[19] *JDK8, http://openjdk.java.net/projects/jdk8/*, accessed 24 January 2014
[20] *Scala, http://www.scala-lang.org/*, accessed 24 January 2014.
[21] Gibbons, J., Design Patterns as Higher-Order Datatype-Generic Programs, *Proc. WGP'06* (2006).
[22] Piponi, D., You Could Have Invented Monads! (And Maybe You Already Have.), *A Neighborhood of Infinity*, http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html, accessed 1 April 2014.
[23] Iverson, K.E., *A Programming Language*, Wiley (1962).