Fission for Program Comprehension

Jeremy Gibbons

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

Abstract. *Fusion* is a program transformation that combines adjacent computations, flattening structure and improving efficiency at the cost of clarity. *Fission* is the same transformation, in reverse: creating structure, ex nihilo. We explore the use of fission for *program comprehension*, that is, for reconstructing the design of a program from its implementation. We illustrate through rational reconstructions of the designs for three different C programs that count the words in a text file.

1 Introduction

Program *fusion* is a meaning-preserving transformation that combines two adjacent computations into one. Those computations might be independent; for example, computing the mean of a sequence of numbers involves both summing and counting the elements of the sequence, and these two independent loops may be fused into one, returning a pair. Alternatively, the computations might be consecutive; for example, testing for membership of a collection can be expressed as comparisons against every element of the collection, then disjoining the results, and these two consecutive loops may be fused into one.

Program fusion is usually seen as an efficiency-improving transformation, perhaps at the cost of comprehensibility. A clear and simple version of a program is developed first, as a composition of strongly coherent but loosely coupled components; for example, membership in terms of comparisons and distributed disjunction, or mean in terms of sum and count. That modular structure might incur unnecessary runtime costs: either in building up an intermediate data structure, only to take it apart straight away, or in making two traversals of a data structure when only one is required. Fusion laws show how to combine components, breaking down the modular structure and the redundant manipulations it entails.

Program *fission* uses the same properties of programs as fusion does, but in the opposite direction. Starting from a complex monolithic optimized program, one constructs a simpler, more modular 'specification' or 'prototype', identifying the components from which the complex program might have been assembled. This construction might be for the first time, for a program that was never properly designed or whose structure has evolved over time from an initial design that has not been kept up to date; or it might be a matter of reconstructing a

lost design. Either way, it can be used for *program comprehension*, that is, for understanding the behaviour of an undocumented unit of code.

Program fission is harder than program fusion, because it entails *entropy reduction*: the introduction of structure, rather than its elimination. It is a fundamental phenomenon of physical systems that entropy increases in a closed system: in order to prevent the inevitable increase in disorder over time, it is necessary to inject some energy into the system. A similar phenomenon seems to arise in logical systems such as software; witness tales of 'software rot', for example [18].

Program fission is one approach among many to the problem of *software re-engineering*, or reconstructing lost or out-of-date documentation for legacy systems. This whole area has been described as being 'about as easy as reconstructing a pig from a sausage' [5]. Indeed, as we shall see, it is harder even than that: a given sausage can have only one explanation, but a given program might have multiple explanations. By analogy, you might not even know that it is a pig you should be reconstructing from your sausage.

2 Notation

We will make use of a Haskell-like notation, for the sake of familiarity; we will also use a number of functions from the Haskell standard library, but we will explain them as we introduce them. However, we will make greater use of sum and product types and less use of currying than is usual in the Haskell language or libraries.

2.1 Sums and products

We use $\alpha \times \beta$ for the product type with first component of type α and second of type β (normally written '(α, β)' in Haskell); the projection functions *fst*, *snd* are as expected. We write '$f \times g$' for the map operation on pairs, applying f to the first component and g to the second, and '$f \vartriangle g$' for the 'fork' operation, taking x to $(f\ x, g\ x)$. The function $twist :: \alpha \times \beta \to \beta \times \alpha$ twists a pair. The unit type is 1 (normally written '()' in Haskell). We also use $\alpha + \beta$ for the sum type (normally written '*Either* α β' in Haskell). In the special case that $\alpha = 1$, we use the injections $Nothing :: 1 + \alpha$ and $Just :: \alpha \to 1 + \alpha$ as in Haskell.

2.2 Datatypes

We will have need of both 'cons lists' (constructed by prefixing elements) and 'snoc lists' (constructed by suffixing). We extend Haskell's neutral notation involving a plain colon for constructing a non-empty list, and use '$\cdot{:}$' for prefixing to a cons list and '${:}\cdot$' for suffixing to a snoc list. The type $[\alpha]$ denotes cons lists with elements of type α, and $\langle\alpha\rangle$ denotes snoc lists. However, we will resort to using the conventional notation '$[\,]$' for the empty list and '$[a]$' for a singleton list in what follows, trusting to context to disambiguate which kind of list is

meant. We will also use a datatype *Nat* of Peano numbers, with $Zero :: Nat$ and $Succ :: Nat \to Nat$.

2.3 Folds

The natural pattern of computation over cons lists, the so-called universal arrow induced by the datatype definition, is called *foldr* in the Haskell library; it consumes list elements, starting at the end of the list. We use the same name here, but give it a slightly different type by uncurrying the binary operator.

$$\begin{aligned} &foldr :: (\alpha \times \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ &foldr\ f\ e\ [\,] \quad\ = e \\ &foldr\ f\ e\ (a :\!\cdot\ x) = f\ (a, foldr\ f\ e\ x) \end{aligned}$$

In contrast, the natural pattern of computation over snoc lists consumes list elements starting from the beginning of the list, since that is how snoc lists are constructed.

$$\begin{aligned} &folds :: (\beta \times \alpha \to \beta) \to \beta \to \langle\alpha\rangle \to \beta \\ &folds\ f\ e\ [\,] \quad\ = e \\ &folds\ f\ e\ (x :\!\cdot\ a) = f\ (folds\ f\ e\ x, a) \end{aligned}$$

The Haskell standard library also provides a variant of *foldr*, which uses an accumulating parameter [2] and consumes the list elements from left to right rather than right to left. Again, we adapt its type.

$$\begin{aligned} &foldl :: (\beta \times \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\ &foldl\ f\ e\ [\,] \quad\ = e \\ &foldl\ f\ e\ (a :\!\cdot\ x) = foldl\ f\ (f\ (e, a))\ x \end{aligned}$$

Note that, apart from the variety of lists, the types of *foldl* and *folds* are identical. Indeed, if we introduce the function $snoc2cons :: \langle\alpha\rangle \to [\alpha]$ to convert from one list type to another, preserving ordering, then for finite cons lists x, it is not difficult to show that

$$folds\ f\ e\ x = foldl\ f\ e\ (snoc2cons\ x)$$

The proof is essentially the same as for Bird and Wadler's *Third Duality Theorem* [4] for *foldl* and *foldr*. (For infinite x, the above result holds only for certain non-strict f.)

2.4 Unfolds

The categorical dual of a fold on lists, which collapses a list to a value, is an unfold, which grows a list from a value. The Haskell standard library provides essentially the right definition for us.

$$\begin{aligned} &unfoldr :: (\beta \to 1 + (\alpha \times \beta)) \to \beta \to [\alpha] \\ &unfoldr\ f\ b = \mathbf{case}\ f\ b\ \mathbf{of} \\ &\qquad\qquad\quad Nothing \quad\ \to [\,] \\ &\qquad\qquad\quad Just\ (a, b') \to a :\!\cdot\ unfoldr\ f\ b' \end{aligned}$$

Here is an analogous operation for growing natural numbers:

$$\begin{aligned}&\mathit{unfoldn} :: (\beta \to 1 + \beta) \to \beta \to \mathit{Nat}\\ &\mathit{unfoldn}\ f\ b = \mathbf{case}\ f\ b\ \mathbf{of}\\ &\qquad\qquad \mathit{Nothing} \to \mathit{Zero}\\ &\qquad\qquad \mathit{Just}\ b' \ \to \mathit{Succ}\ (\mathit{unfoldn}\ f\ b')\end{aligned}$$

2.5 Paramorphisms and hylomorphisms

Meertens [14] presents a generalization of folds called *paramorphisms*, which correspond to the primitive recursive definitions. Practically, these are characterized by having available, as well as the results of recursive calls, the data substructures on which those calls were made. We will use the paramorphism operator for snoc lists:

$$\begin{aligned}&\mathit{paras} :: ((\beta \times \langle\alpha\rangle) \times \alpha \to \beta) \to \beta \to \langle\alpha\rangle \to \beta\\ &\mathit{paras}\ f\ e\ [\,] \qquad = e\\ &\mathit{paras}\ f\ e\ (x :\cdot a) = f\ ((\mathit{paras}\ f\ e\ x, x), a)\end{aligned}$$

Meijer, Fokkinga and Paterson [15] introduce what they call a *hylomorphism*, which is the composition of an unfold (to generate a data structure) and a fold (to consume that data structure). We use the cons list instance:

$$\begin{aligned}&\mathit{hylor} :: (\alpha \to 1 + (\beta \times \alpha)) \to (\beta \times \gamma \to \gamma) \to \gamma \to \alpha \to \gamma\\ &\mathit{hylor}\ g\ f\ e = \mathit{foldr}\ f\ e \circ \mathit{unfoldr}\ g\end{aligned}$$

The crucial fact about hylomorphisms is that the intermediate data structure is a virtual data structure [19], and (under certain mild strictness conditions) may be *deforested* [21]. In our case, this gives:

$$\begin{aligned}&\mathit{hylor}\ g\ f\ e\ a = \mathbf{case}\ g\ a\ \mathbf{of}\\ &\qquad\qquad \mathit{Nothing} \quad \to e\\ &\qquad\qquad \mathit{Just}\ (b, a') \to f\ (b, \mathit{hylor}\ g\ f\ e\ a')\end{aligned}$$

2.6 Fusion

Each of the various recursion patterns introduced above enjoys a crucial property called *fusion*, whereby an adjacent computation can be absorbed. We will use the fusion laws for *folds*:

$$h \circ \mathit{folds}\ f\ e = \mathit{folds}\ g\ (h\ e) \ \Leftarrow\ h \circ f = g \circ (h \times \mathit{id})$$

and for *paras*:

$$h \circ \mathit{paras}\ f\ e = \mathit{paras}\ g\ (h\ e) \ \Leftarrow\ h \circ f = g \circ ((h \times \mathit{id}) \times \mathit{id})$$

To be precise, each of these fusion laws has mild side conditions concerning strictness, but we elide them here because they do not affect subsequent calculations.

For more details, including proofs of the fusion laws from universal properties of the recursion patterns, see for example [8].

2.7 Functors

Finally, many datatypes form *functors*, operations on types with a corresponding 'map' operation on functions:

$$fmap :: Functor\ f \Rightarrow (\alpha \to \beta) \to f\ \alpha \to f\ \beta$$

We will use this just for the 'maybe' functor taking α to $1 + \alpha$.

3 Counting words

We will use as an illustration in this paper the Unix word count utility `wc`, a now-standard example in the program comprehension literature. The program shown in Figure 1 is taken from Kernighan and Ritchie's classic book on the C programming language [13], and counts the characters, words and lines in a text file. In fact, it is really only the word counting aspect of this program that

```
#include <stdio.h>
#define IN   1  /* inside a word */
#define OUT  0  /* outside a word */

/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

Fig. 1. Kernighan and Ritchie's `wc` program

has interesting structure; counting the characters is simply computing the length of the text, and counting the lines is implemented as counting the newline characters, which is the length of the text filtered for newlines. So we will actually

```
#include <stdio.h>
#define IN   1  /* inside a word */
#define OUT  0  /* outside a word */

int blank(int c) {
  return ( c==' ' || c=='\n' || c=='\t');
}

/* count words in input */
main()
{
    int c, nw, state;
    state = OUT;
    nw = 0;
    while ((c = getchar()) != EOF) {
        if (blank(c))
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d\n", nw);
}
```

Fig. 2. The word-counting slice of Kernighan and Ritchie's `wc` program

start with the program in Figure 2, which counts only the words. This might be considered as the first step in re-engineering a specification from the original program, by *slicing* that program into three independent aspects [22, 20]. (Indeed, slicing is a fission transformation, reversing the fusion of independent but similarly-structured computations.)

We argue that the C program in Figure 2 is 'obviously' equivalent to the following functional program. The imperative loop has been converted to a tail-recursive function.

$$
\begin{aligned}
&wc_1 :: [Char] \to Integer \\
&wc_1 = fst \circ foldl\ step_1\ (0, False) \\
&step_1\ ((n, b), \quad c) \mid blank\ c = (n, False) \\
&step_1\ ((n, True), c) \qquad\quad = (n, True) \\
&step_1\ ((n, False), c) \qquad\quad = (n + 1, True) \\
&blank\ c = (c == \texttt{' '}) \vee (c == \texttt{'\textbackslash n'}) \vee (c == \texttt{'\textbackslash t'})
\end{aligned}
$$

Characters come from a string argument rather than standard input, and the count is returned as an integer result rather than printed to standard output. In a fuller study of program comprehension, one would make this equivalence between imperative and functional programming more explicit; but for our purposes —

namely, illustrating program fission — we will take the functional program wc_1 as the starting point.

In the remainder of this paper, we reconstruct a number of different implementations $wc_2, wc_3 ...$ of wc_1. They will all be extensionally equal, possibly modulo different representations of lists, but will have different structures and hence different 'explanations'. We refer to this common behaviour collectively as wc.

4 Directionality

The first observation we make about program wc_1 is that tail recursion — here, in the form of a *foldl* — is somewhat alien to a lazy functional programmer, although — in the form of a `while` loop — it comes quite naturally to an imperative programmer. Program wc_1 would be more comprehensible if it were expressed in a less alien idiom.

With the benefit of understanding of the purpose of the program, namely that it counts words, we could reasonably argue that it does not matter whether we scan the input from left to right or vice versa. We could therefore refactor wc as follows:

$$\begin{aligned} wc_2 &= fst \circ foldr\ step_2\ (0, False) \\ step_2 &= step_1 \circ twist \end{aligned}$$

However, there are two counter-arguments. The first is that, although this refactoring seems reasonable, its formal justification is not so obvious. In particular, it is not the case that the uses of *foldl* in wc_1 and *foldr* in wc_2 are equal: a text that starts with a non-blank but ends with a blank will yield different boolean state values in different directions, even though the number of words is the same both ways. The second counter-argument is that this step depends on understanding the purpose of the program, which is exactly what is unavailable in a program comprehension exercise.

So we take an alternative approach: adapt the underlying data structure to reflect more closely the pattern of computation. After all, we introduced the list type into the problem in the first place: it was not present in the C program. Specifically, the left-to-right traversal in the C program is the natural pattern of computation on snoc lists rather than on cons lists. Therefore, in place of the tail-recursive *foldl* pattern for cons lists, we use the naturally recursive fold on snoc lists:

$$folds :: (\beta \times \alpha \to \beta) \to \beta \to \langle \alpha \rangle \to \beta$$

We therefore make the following refactoring instead.

$$\begin{aligned} wc_3 &:: \langle Char \rangle \to Integer \\ wc_3 &= fst \circ folds\ step_1\ (0, False) \end{aligned}$$

We might elevate this step to a general principle of program comprehension: *consider carefully the data structures used, because they determine the pattern of*

computation. This strengthens the case for functional programming as a medium for program comprehension; functional programming encourages the definition of tailor-made datatypes, rather than shoe-horning a problem into one of a fixed collection of general-purpose but sometimes ill-fitting datatypes.

5 Extracting length

Now, it seems reasonable (although still an invention) that *wc* involves *length* somehow; in particular, *wc* is *length* after some initial computation. It is evident that wc_3 is counting something, because the result that is returned is constructed from an initial zero that is occasionally incremented. Perhaps we could elevate this observation to a second principle of program comprehension: *if a program returns a count, investigate what it is counting.*

Returning to the program, we name the generator of things to be counted *words* — without any justification as yet, but what's in a name?

$$wc_4 = length \circ words_4$$

Since wc_4 should equal wc_3, in order to deduce a definition of $words_4$, we need to extract a factor of *length* from the definition of wc_3. Extracting this factor from the *fst* is straightforward, since, by the pair calculus,

$$fst \circ f = length \circ fst \circ g \;\Leftarrow\; f = (length \times id) \circ g$$

That is, extracting a factor of *length* from wc_3 amounts to extracting a factor of $length \times id$ from $folds\ step_1\ (0, False)$.

Now we can use fission — fusion in reverse — to deduce e and $glue_4$ (the latter another name chosen with hindsight) such that

$$folds\ step_1\ (0, False) = (length \times id) \circ folds\ glue_4\ e$$

For the seed, this requires $(length \times id)\ e = (0, False)$, and so $e = ([\,], False)$. For the binary operator, it requires

$$(length \times id) \circ glue_4 = step_1 \circ ((length \times id) \times id)$$

This equation characterizes a data refinement relation between $glue_4$ and $step_1$, where $glue_4$ is the abstract operation, $step_1$ the concrete operation, and $length \times id$ the abstraction function. Informally, where $glue_4$ trades in sequences, $step_1$ trades in their lengths. Normally, however, one uses a data refinement relationship to derive a concrete implementation from an abstract one; here, of course, we need to go in the opposite direction.

That is, where $step_1$ trades in numbers, we need to construct a function $glue_4$ that trades in their 'unlengths', or sequences of those lengths. Of course, 'unlength' is not a function; there are many sequences of a given length. As a consequence, the data refinement relationship does not completely determine $glue_4$. We need to exercise some creativity in inventing suitable sequences of given lengths. It seems reasonable to suggest that we should use as little creativity as

possible in inventing such sequences. This corresponds in our physical analogy with entropy to minimizing the energy injected into the system. Moreover, one might expect that the less creative we are at any given step in a re-engineering exercise, the more freedom there is in later steps (and the less likely we are to lead ourselves into a dead end).

For example, the first clause of the definition of $step_1$ entails that, when c is blank, $glue_4\ ((ws, b), c)$ should be a pair $(ws', False)$ such that ws and ws' have the same length. The least creative way to achieve this is naturally to let $ws' = ws$.

$$glue_4\ ((ws, b), c) \mid blank\ c = (ws, False)$$

For the second clause, when c is non-blank, we require $glue_4\ ((ws, True), c)$ to be a pair $(ws', True)$ where ws should again be the same length ws'. We could do the same thing again, equating the two sequences, but in fact there is an even less creative way of proceeding. We note that the physical theory of information states that it requires energy to erase data as well as to invent it. Therefore, we look for a way to use c, combining it with ws while maintaining the latter's length. This is straightforward to do, if ws is a sequence of sequences of characters, provided that it is non-empty: we suffix c to the last sequence in ws. Fortunately, it is an invariant of the fold in wc_3 that when the boolean component of the pair is $True$, the integer component is greater than zero, so our abstract value ws will be a non-empty sequence.

$$glue_4\ ((ws :\cdot w, True), c) = (ws :\cdot (w :\cdot c), True)$$

Finally, for the third clause, again we assume that c is non-blank, and we require $glue_4\ ((ws, False), c)$ to be a pair $(ws', False)$ where ws' is one longer than ws. The least creative way to extend the sequence of strings ws by one string, using the given data c, is to suffix c as an additional singleton string.

$$glue_4\ ((ws, False), c) = (ws :\cdot [c], True)$$

Assembling these three cases, we have

$$\begin{aligned} &glue_4\ ((ws, b), c) \mid blank\ c = (ws, False) \\ &glue_4\ ((ws :\cdot w, True), c) \quad = (ws :\cdot (w :\cdot c), True) \\ &glue_4\ ((ws, False), c) \qquad\ = (ws :\cdot [c], True) \end{aligned}$$

And to rewind the reasoning that led us here: if we let

$$words_4 = fst \circ folds\ glue_4\ ([\,], False)$$

then indeed the composition

$$wc_4 = length \circ words_4$$

computes the words in a text, and proceeds to count them.

6 Mind the gap

In Section 5, we used an argument based on entropy to suggest that, when introducing structure in order to satisfy a fusion law, one should both invent and discard as little information as possible. We stuck to this principle for the second and third clauses of the function $glue_4$, but wavered a little in our resolve when it came to the first clause. The function $words_4$ does discard some information that might be preserved: it conflates different blank characters, such as spaces and newlines, and it also fails to keep track of how many blanks separate words. Therefore, $words_4$ is not invertible. How would the development have proceeded if we had stuck to our principle, and found a way to preserve blank characters?

In the case that c is blank, we wanted $glue_4\ ((ws, b), c)$ to return a pair $(ws', False)$ such that ws and ws' have the same length. We chose to let $ws' = ws$, but this required us to discard c. We can preserve c while maintaining the length of the first component of the pair, provided that that first component is non-empty:

$$glue\ ((ws :\cdot w, b), c) \mid blank\ c = (ws :\cdot (w :\cdot c), False)$$

In effect, this corresponds to representing each word in the input as a non-empty sequence of non-blanks followed by a (possibly-empty, in the case of the last word) sequence of blanks. However, there is nowhere to keep the c while preserving the length of an empty first component — because this representation does not capture blanks at the start of the input. We therefore augment the state, the result of *words*, to represent also the possibly-empty sequence of blanks at the start of the input.

$$\begin{aligned}
&words_5 :: \langle Char\rangle \to \langle Char\rangle \times \langle\langle Char\rangle\rangle\\
&words_5 = fst \circ folds\ glue_5\ (([\,], [\,]), False)\\
&wc_5 = (length \circ snd) \circ words_5\\
&glue_5\ (((wb, [\,]), b), c) \qquad \mid blank\ c = ((wb :\cdot c, [\,]), False)\\
&glue_5\ (((wb, ws :\cdot w), b), c) \mid blank\ c = ((wb, ws :\cdot (w :\cdot c)), False)\\
&glue_5\ (((wb, ws :\cdot w), True), c) \qquad = ((wb, ws :\cdot (w :\cdot c)), True)\\
&glue_5\ (((wb, ws), False), c) \qquad = ((wb, ws :\cdot [c]), True)
\end{aligned}$$

Note now that the boolean component of the state is redundant, as it can be determined from the remaining components.

7 A different starting point

The Kernighan and Ritchie C programs above maintain in their main loops, in addition to the counts which are the point of the exercise and will eventually be printed out, a boolean variable `state` indicating whether, if the next character to be read is a non-blank, it will start a new word. One might start with a different program: one that dispenses with this boolean variable, but uses instead a two-character window onto the text to determine which non-blank characters start words. Such a program is shown in Figure 3. We omit the definition of `blank`,

```
#include <stdio.h>

/* count words in input */
main()
{
  int c, d, nw;
  nw = 0;
  d = ' ';
  c = getchar();
  while (c != EOF) {
    if (!blank(c) && blank(d)) {
      ++nw;
    }
    d = c; c = getchar();
    }
  printf("%d\n", nw);
}
```

Fig. 3. A different `wc` program, with a two-character window

since it is identical to the one given earlier. In this section, we subject this new program to the same kind of reasoning as before, to determine whether it could be considered as having 'the same explanation' but with a different implementation, or whether it really arose from a different design.

The program in Figure 3 maintains the invariant that the variable `d` records the character before the 'current' character `c` (except initially, when it acts as a space character). This is an instance of the general paramorphism pattern, whereby the treatment of each element depends not just on the treatment of previous elements, but also on those previous elements themselves.

$$
\begin{array}{ll}
wc_6 = paras\ step_6\ 0 & \\
step_6\ ((n, x), c) & \mid blank\ c = n \\
step_6\ ((n, x :\cdot d), c) & \mid blank\ d = n + 1 \\
step_6\ ((n, x :\cdot d), c) & = n \\
step_6\ ((n, [\,]), c) & = n + 1
\end{array}
$$

Note that the use of the paramorphism pattern encodes the invariant that $n = wc_6\ x$ in every application $step_6\ ((n, x), c)$. It therefore explicitly captures the invariant about the value of variable `d`, which must therefore be comprehended from the code. It also provides a separate initial boundary condition to remove the need for the 'virtual' space character before the first 'real' character.

As before, we try to write this as the composition of *length* with some simpler function, using paramorphism fission. Clearly, the seed of the paramorphism has to be $[\,]$, the only sequence with length 0. For the operator, the fusion condition is that

$$length \circ glue_7 = step_6 \circ ((length \times id) \times id)$$

If we can construct a function $glue_7$ satisfying this condition, then $length$ fuses with $paras\ glue_7\ [\,]$ to give wc_6.

For the first clause, when c is blank, apparently $glue_7$ should simply return the first of its three argument components.

$$glue_7\ ((ws, x), c) \mid blank\ c = ws$$

For the second clause, when c is non-blank, the initial segment $x :\cdot d$ of characters seen so far is non-empty, and the previous character d is blank, we should return a sequence one longer than the first argument component ws. The obvious thing to do is to suffix a new element to ws, and the least creative data-preserving way of doing that is to suffix $[c]$.

$$glue_7\ ((ws, x :\cdot d), c) \mid blank\ d = ws :\cdot [c]$$

For the third clause, when c is non-blank, the initial segment $x :\cdot d$ is non-empty, and d is also non-blank, we should return a sequence the same length as the first argument component ws. Returning ws unchanged loses the data c. A less creative way would be to preserve c by combining it with data in ws, provided the latter is non-empty. Fortunately, it is an invariant that in this circumstance ws is non-empty.

$$glue_7\ ((ws :\cdot w, x :\cdot d), c) = ws :\cdot (w :\cdot c)$$

For the fourth and final clause, when c is non-blank but the initial segment of the list is empty, we need to extend the sequence by a single element. The least creative type-correct way to do this is make a singleton string from c.

$$glue_7\ ((ws, [\,]), c) = ws :\cdot [c]$$

Summing up, we have deduced the following definition of $glue_7$:

$$\begin{array}{lll} glue_7\ ((ws, x), c) & \mid blank\ c & = ws \\ glue_7\ ((ws, x :\cdot d), c) & \mid blank\ d & = ws :\cdot [c] \\ glue_7\ ((ws :\cdot w, x :\cdot d), c) & & = ws :\cdot (w :\cdot c) \\ glue_7\ ((ws, [\,]), c) & & = ws :\cdot [c] \end{array}$$

We then define

$$\begin{array}{lll} wc_7 & = length \circ words_7 \\ words_7 & = paras\ glue_7\ [\,] \end{array}$$

Using paramorphism fusion, the $length$ combines with the paramorphism, yielding the earlier program wc_6. Moreover, $words_7$ does indeed yield the individual words in the text.

8 Nested loops

All the C programs for the wordcount problem that we have seen so far have a single loop, with additional hidden state to determine the behaviour of the loop

```
#include <stdio.h>

main() {
  int c=getchar(), nw=0;
  while (1) {
    while (c != EOF && blank(c))
        c=getchar();
    if (c == EOF)
        break;
    nw++;
    while (c != EOF && !blank(c))
        c=getchar();
  }
  printf("%d\n", nw);
}
```

Fig. 4. A `wc` program with nested loops

body. A different way of solving the problem is to use nested loops, in effect using the program counter instead of that hidden state. One such program is shown in Figure 4. In this program, the variable `c` always contains the next character in the text, or the `EOF` character at the end of the text. The outer loop runs indefinitely. The first inner loop skips blanks. If this first inner loop reaches the end of the text, control breaks out of the outer loop and the program quits. Otherwise, the first inner loop terminated because it reached a non-blank character; the number of words is incremented, and the rest of that word skipped.

We claim that this program has the following 'obvious' functional equivalent.

$$\begin{aligned} wc_8\ x = \mathbf{let}\ & y = dropWhile\ blank\ x\ \mathbf{in} \\ & \mathbf{if}\ null\ y\ \mathbf{then}\ 0 \\ & \quad \mathbf{else}\ \ 1 + wc_8\ (dropWhile\ (not \circ blank)\ y) \end{aligned}$$

where *null* is the predicate that returns *True* precisely of the empty list, and $dropWhile :: (\alpha \to Bool) \to [\alpha] \to [\alpha]$ takes a predicate p and a list x and discards the longest prefix of x all of whose elements satisfy p. (Strictly speaking, getting to this point entails the elimination of the accumulating parameter that is the word count.) This program matches the pattern of a list hylomorphism:

$$\begin{aligned} & wc_9 :: [Char] \to Integer \\ & wc_9 = hylor\ word_9\ plus_9\ 0 \\ & plus_9\ (w, n) = 1 + n \\ & word_9\ x = \mathbf{let}\ y = dropWhile\ blank\ x\ \mathbf{in} \\ & \qquad \mathbf{if}\ null\ y\ \mathbf{then}\ Nothing \\ & \qquad\quad \mathbf{else}\ \ Just\ ((), dropWhile\ (not \circ blank)\ y) \end{aligned}$$

Of course, a hylomorphism fissions automatically into a fold after an unfold:

$$wc_{10} = foldr\ plus_9\ 0 \circ unfoldr\ word_9$$

And as expected, the fold phase is just *length*, and counts the items generated by the unfold phase; these items are all units, but there is precisely one of them for each word.

We might apply the principle of least creativity again, preserving those non-blank elements of y discarded by $word_9$:

$$\begin{aligned} word_{11}\ x = \mathbf{let}\ y = dropWhile\ blank\ x\ \mathbf{in} \\ \quad \mathbf{if}\ null\ y\ \mathbf{then}\ Nothing \\ \quad\quad \mathbf{else}\ \ Just\ (span\ (not \circ blank)\ y) \end{aligned}$$

Here, $span :: (\alpha \to Bool) \to [\alpha] \to [\alpha] \times [\alpha]$ is a generalization of $dropWhile$: it takes a predicate p and a list x and returns a pair of lists (y, z) such that $y +\!\!+ z = x$ and $y = dropWhile\ p\ x$.

When $hylor\ word_{11}\ plus_9\ 0$ is fissioned, we get:

$$wc_{11} = foldr\ plus_9\ 0 \circ unfoldr\ word_{11}$$

Here, the unfold phase really is just *words* again.

It might seem curious that we have reverted to cons lists for the virtual data structure of the hylomorphism, rather than continuing to work with snoc lists. But of course, the virtual data structure of a hylomorphism merely encapsulates the pattern of recursion, and hylomorphisms for cons lists and snoc lists are entirely equivalent.

9 Counting revisited

Let us return our attention to the recursive equivalent of the program with nested loops from Figure 4:

$$\begin{aligned} wc_8\ x = \mathbf{let}\ y = dropWhile\ blank\ x\ \mathbf{in} \\ \quad \mathbf{if}\ null\ y\ \mathbf{then}\ 0 \\ \quad\quad \mathbf{else}\ \ 1 + wc_8\ (dropWhile\ (not \circ blank)\ y) \end{aligned}$$

Our reconstruction in Section 8 started from the observation that this recursive program is an instance of the hylomorphism pattern on lists. However, the list algebra involved in this hylomorphism is a very special one, namely the initial algebra of natural numbers. This leads to another explanation of the same program.

We adapt the type of the function, so that it returns a recursively-constructed natural number rather than a built-in integer.

$$\begin{aligned} wc_{12}\ \ &:: [Char] \to Nat \\ wc_{12}\ x = \mathbf{let}\ y = dropWhile\ blank\ x\ \mathbf{in} \\ \quad \mathbf{if}\ null\ y\ \mathbf{then}\ Zero \\ \quad\quad \mathbf{else}\ \ Succ\ (wc_{12}\ (dropWhile\ (not \circ blank)\ y)) \end{aligned}$$

Now we see immediately that this is a straightforward instance of *unfoldn*:

$$\begin{aligned} &wc_{13} = unfoldn\ dropWord_{13} \\ &dropWord_{13}\ x = \mathbf{let}\ y = dropWhile\ blank\ x\ \mathbf{in} \\ &\qquad\qquad \mathbf{if}\ null\ y\ \mathbf{then}\ Nothing \\ &\qquad\qquad\qquad\qquad \mathbf{else}\ \ Just\ (dropWhile\ (not \circ blank)\ y) \end{aligned}$$

In fact, this observation is an instance of a more general rule about the composition of *length* and an unfold to lists:

$$length \circ unfoldr\ f = nat2int \circ unfoldn\ (fmap\ snd \circ f)$$

where $nat2int :: Nat \to Integer$ coerces from recursively-constructed naturals to built-in integers. This law could be phrased as a principle of counting: rather than enumerating a list of things, then computing the length of that list, one can more directly count the number of times the operation 'discard a thing' can be performed.

Unfolds to the naturals are surprisingly common, despite the unfamiliarity of the operator *unfoldn* itself. The law above suggests that they capture many counting problems. Gibbons [9] shows that *unfoldn* is essentially the minimization operator from recursive function theory, the additional operator needed to progress from the primitive recursive to the general recursive functions, or equivalently from `for` to `while` loops. For example, integer division is an unfold to naturals, since dividing by m is the same as computing the number of times m can be subtracted without the difference becoming negative. Elsewhere [11] we have argued that even unfolds to lists are underappreciated; we believe that argument applies a fortiori to unfolds to other datatypes such as the naturals.

10 Discussion

The reconstruction of specifications from programs is an important part of a larger endeavour called *software renovation*. This field addresses the difficult problem of maintaining legacy software when its design documentation is unavailable: it might have become out of date, or been lost altogether, or it might never have existed in the first place. In order to modify undocumented software, one essentially is forced to spend some effort in comprehending the existing system (unless one is prepared to use trial and error, making random changes and hoping for a useful result). *Program comprehension* might be as lightweight as simply attempting to understand one small module of code and its interface in isolation, or it might involve retracing one's steps all the way back towards a high-level design for the entire system, or some level in between. However much ones tries to comprehend, one works backwards from implementation to design, with the aim of modifying that design and working forwards again to a revised implementation.

The view we have taken in this paper is that the essence of a design is expressed in terms of higher-order recursion patterns. A similar view underlies our and others' arguments [9, 1] that the different designs for sorting algorithms

embodied by insertion sort, merge sort, quick sort and so on arise from using different patterns of recursion. If one accepts the claim that design patterns in object-oriented programming correspond to recursion patterns in generic functional programming [10], then this is further support for Johnson's slogan that 'patterns document architectures' [12].

An advantage of using a formal linguistic vehicle such as functional programming for expressing patterns, rather than the informal prose and pictures that is traditional in the patterns community [7], is that those patterns and the programs that exhibit them may be manipulated and reasoned about mathematically. In particular, well-known *fusion laws* can be used to flatten the structure imposed by a pattern, for efficiency; in this paper, we have used those laws in reverse as *fission laws* in order to recover lost structure.

We have examined three different C programs for counting the words in a text file, and attempted to reverse engineer designs from these implementations. Naturally, different implementations of a program arise from different designs for those implementations; but it is reasonable to ask how divergent those designs are: how much of the development is shared, and how late in the process do the evolutionary forks appear?

In fact, we have shown that the three different wordcount programs might all have arisen from the same high-level design, namely the composition $length \circ words$. The differences between the three programs are explained in terms of different strategies for implementing *words*: as a fold, a paramorphism, or an unfold — the first two of which are inductive, the last coinductive. However, the coinductive design lends itself to an alternative explanation of the problem, in terms of counting rather than enumeration, which might be considered a second high-level design.

Acknowledgements

This paper was inspired by a discussion with José Nuno Oliveira at the 59th meeting of IFIP Working Group 2.1 in Nottingham in September 2004. The use of the Unix `wc` utility as a standard benchmark example in the program comprehension community appears to be due to Gallagher and Lyle [6], although we learnt about it from Oliveira [16]. Our approach has been informed by Oliveira's 'Program Understanding and Re-engineering' project [17].

We are indebted to a number of people who have helped to improve this paper. We are grateful to the participants at the PURe workshop in Minho in October 2005, especially to José Nuno Oliveira for the invitation to speak, and to Alcino Cunha for the realisation that our program wc_8 is really an unfold to the naturals. Members of the Algebra of Programming research group at Oxford, and of the Datatype-Generic Programming project, made a number of insightful comments; the quip about not even knowing it is a pig you should reconstruct is due to Geraint Jones. The code is formatted with Ralf Hinze and Andres Löh's wonderful lhs2TEX translator.

References

1. Lex Augusteijn. Sorting morphisms. In S. D. Swierstra, P. R. Henriques, and J. N. Oliveira, editors, *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 1–27, 1998.
2. Richard S. Bird. The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, 6(4):487–504, October 1984. See also [3].
3. Richard S. Bird. Addendum to "The promotion and accumulation strategies in transformational programming". *ACM Transactions on Programming Languages and Systems*, 7(3):490–492, July 1985.
4. Richard S. Bird and Philip L. Wadler. *An Introduction to Functional Programming*. Prentice-Hall, 1988.
5. Allen Eastwood. It's a hard sell — and hard work too (software re-engineering). *Computing Canada*, 18(22):35, 1992.
6. K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. 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.
9. Jeremy Gibbons. Origami programming. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 41–60. Palgrave, 2003.
10. Jeremy Gibbons. Patterns in datatype-generic programming. In Jörg Striegnitz and Kei Davis, editors, *Multiparadigm Programming*, volume 27. John von Neumann Institute for Computing (NIC), 2003. Proceedings of the First International Workshop on Declarative Programming in the Context of Object-Oriented Languages (DPCOOL).
11. 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.
12. Ralph Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming: Systems, Languages and Applications*, Vancouver, 1992.
13. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.
14. Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
15. 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.
16. José Nuno Oliveira. Bagatelle in C arranged for VDM SoLo. *Journal of Universal Computer Science*, 7(8):754–781, 2001. Formal Aspects of Software Engineering: Special Issue in honour of Peter Lucas.
17. Program understanding and re-engineering: Calculi and applications. Web site, `http://wiki.di.uminho.pt/wiki/bin/view/PURe/`, 1999-2005.
18. Eric S. Raymond, editor. *The New Hacker's Dictionary*. MIT Press, 1991.

19. Doaitse Swierstra and Oege de Moor. Virtual data structures. 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 355–371. Springer-Verlag, 1993.
20. Gustavo Villavicencio and José Nuno Oliveira. Reverse program calculation supported by code slicing. In *Eighth Working Conference on Reverse Engineering*, pages 35–48. IEEE, 2001.
21. Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
22. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.